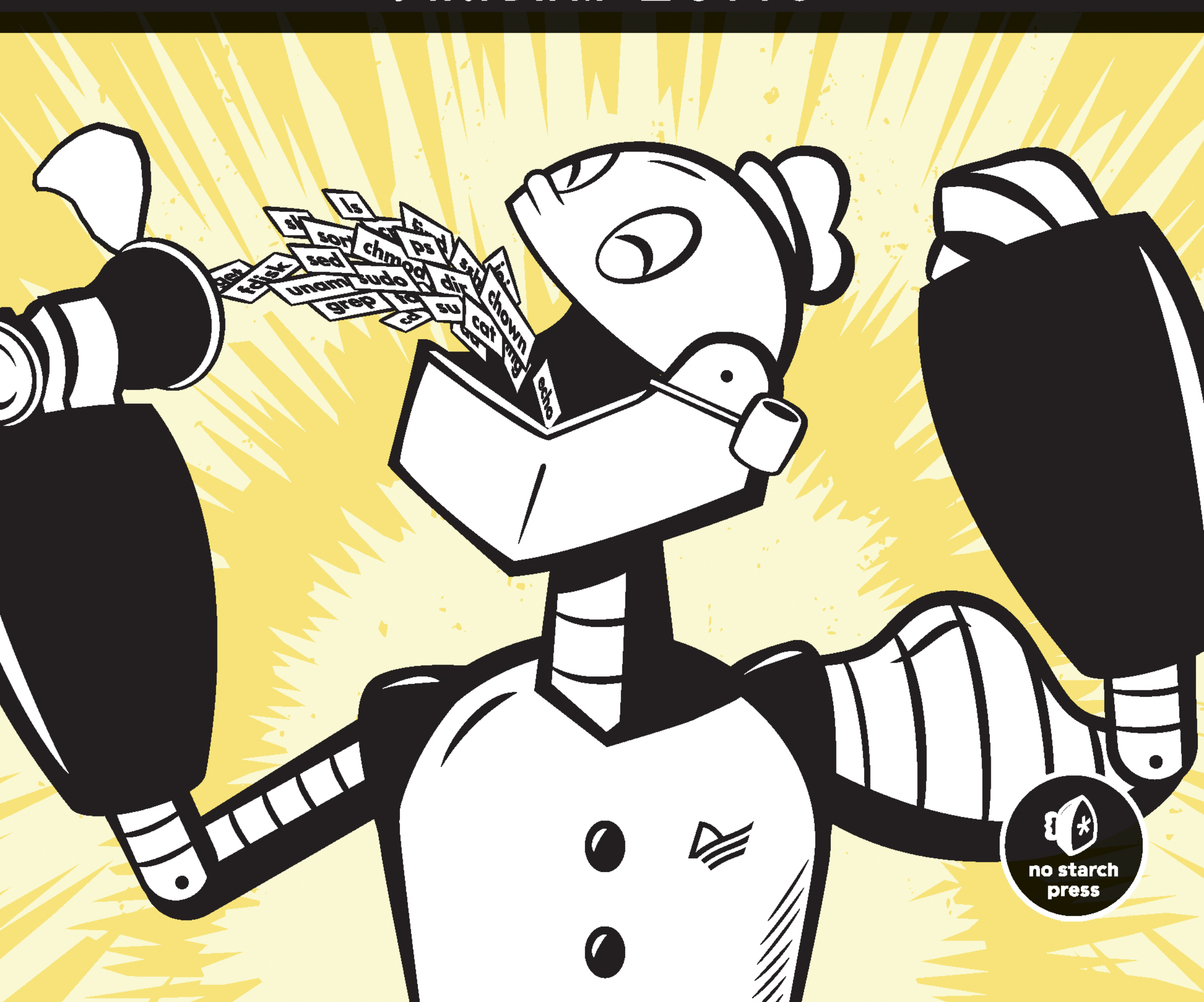
2-Е МЕЖДУНАРОДНОЕ ИЗДАНИЕ

КОМАНДНАЯ СТРОКА LINUX

ПОЛНОЕ РУКОВОДСТВО

УИЛЬЯМ ШОТТС



by William Shotts

THE LINUX COMMAND LINE

2ND EDITION

A Complete Introduction



no starch press
San Francisco

УИЛЬЯМ ШОТТС

КОМАНДНАЯ СТРОКА LINUX

ПОЛНОЕ РУКОВОДСТВО

2-Е МЕЖДУНАРОДНОЕ ИЗДАНИЕ







Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2020

ББК 32.973.2-018.2
УДК 004.451
Ш80

Шоттс У.

Ш80 Командная строка Linux. Полное руководство. 2-е межд. изд. — СПб.: Питер, 2020. — 544 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1430-6

Международный бестселлер «Командная строка Linux» поможет преодолеть путь от первых робких щелчков по клавишам до уверенного создания полноценных программ для последней версии bash — наиболее популярной командной оболочки Linux.

Второе издание рассказывает о новых возможностях bash 4.x, таких как новые операторы перенаправления и операции подстановки. В разделе, посвященном сценариям оболочки, рассматриваются современные надежные методы программирования, способы предотвращения типичных ошибок и потенциально опасных ситуаций.

Вы освоите неустаревающие навыки владения командной строкой: навигацию по файловой системе, настройку окружения, объединение команд в цепочки и сопоставление с регулярными выражениями. Вы постигнете философию, лежащую в основе многих инструментов командной строки, разберетесь с богатым наследием, полученным от суперкомпьютеров с Unix, и приобщитесь к знаниям, накопленным поколениями гуру, исключивших мышь из своего арсенала инструментов.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.2
УДК 004.451

ISBN 978-1593279523 англ.

ISBN 978-5-4461-1430-6

Краткое содержание

Часть III. ТИПИЧНЫЕ ЗАДАЧИ И ОСНОВНЫЕ ИНСТРУМЕНТЫ

Часть IV. СЦЕНАРИИ КОМАНДНОЙ ОБОЛОЧКИ

Оглавление



Посвящается Карен

Об авторе

Уильям Шоттс (William Shotts) — профессиональный разработчик программного обеспечения с более чем 30-летним стажем, который уже больше 20 лет активно пользуется операционной системой Linux. Имеет богатый опыт разработки программного обеспечения, технической поддержки, контроля качества и написания документации. Также является создателем LinuxCommand.org, образовательного и информационно-просветительского сайта, посвященного Linux, где публикуются новости, обзоры и оказывается поддержка пользующимся командной строкой Linux.

О научном редакторе

Жорди Гутьеррес Эрмосо (Jordi Gutiérrez Hermoso) — программист, математик и сторонник этичного хакерства. С 2002 года пользуется исключительно Debian GNU/Linux не только дома, но и на работе. Жорди участвует в разработке GNU Octave, бесплатной вычислительной среды, во многом совместимой с Matlab, а также Mercurial, распределенной системы управления версиями. Увлекается чистой и прикладной математикой, катанием на коньках, плаванием и вязанием. В последнее время много думает о проблемах выброса парниковых газов и участвует в акциях по спасению носорогов.

Благодарности

Я хочу поблагодарить всех, кто помог появиться на свет этой книге.

К первому изданию

В первую очередь хочу поблагодарить тех, кто вдохновил меня:

Дженни Уотсон (Jenny Watson), рецензента издательства Wiley Publishing, — она первая предложила написать книгу о языке сценариев командной оболочки.

Джона Дворака (John C. Dvorak), известного колумниста и большого эрудита, — в эпизоде своего видеоподкаста «Cranky Geeks» он дал великолепный посыл творческому процессу: «Черт! Пишите по 200 слов в день в течение года, и получите роман». Следуя его совету, я писал по одной странице в день до тех пор, пока не получил книгу.

Дмитрия Попова (Dmitri Popov), написавшего статью в журнале «Free Software Magazine» под названием «Creating a book template with Writer», — именно она вдохновила меня использовать OpenOffice.org Writer (позднее переименованный в LibreOffice Writer) для набора текста. Результат получился изумительным.

Далее, спасибо добровольцам, которые помогли выпустить оригинал, свободно распространяемую версию этой книги (доступна на LinuxCommand.org):

Марка Полески (Mark Polesky), который выполнил большую редакторскую работу и проверил текст книги.

Джесси Беккер (Jesse Becker), Томаш Хщонович (Tomasz Chrzczonowicz), Майкл Левин (Michael Levin) и Спенс Майнер (Spence Miner) также проверили отдельные фрагменты книги и представили свои рецензии.

Карен М. Шоттс (Karen M. Shotts) много часов посвятила правке моей оригинальной рукописи.

Наконец, спасибо добрым людям из No Starch Press, которые серьезно потрудились над созданием коммерческой версии книги:

Серене Янг (Serena Yang), управляющей производством; Киту Фанчеру (Keith Fancher), моему редактору; и остальным сотрудникам No Starch Press.

Ко второму изданию

Особое спасибо добровольцам, приславшими свои ценные отзывы о свободно распространяемой версии книги и ко второму изданию, выпущенному No Starch Press: Адриану Арпидесу (Adrian Arpidez), Ху Бо (Hu Bo), Стиву Брэггу (Steve Bragg), Джону Барнсу (John Burns), Паоло Казати (Paolo Casati), Хериберто Канту (Heriberto Cantú), Энцо Кардиналу (Enzo Cardinal), Ликсину Дуану (Lixin Duan), Джошуа Эскамилле (Joshua Escamilla), Брюсу Фаулеру (Bruce Fowler), Двину Харперу (Devin Harper), Йоргену Хейтманну (Jørgen Heitmann), Джонатану Джонсу (Jonathan Jones), Сунил Джоши (Sunil Joshi), Ма Джун (Ma Jun), Эрику Каммереру (Eric Kammerer), Сету Кингу (Seth King), Крису Найту (Chris Knight), Ярославу Колосовскому (Jaroslaw Kolosowski), Джиму Ковачу (Jim Kovacs), Бартломею Майку (Bartłomiej Majka), Башару Мари (Bashar Maree), Френку Мактипсу (Frank McTipps), Майку О'Доннеллу (Mike O'Donnell), Джастину Пейджу (Justin Page), Парвизу Расулипуру (Parviz Rasoulipour), Уолдо Рибейро (Waldo Ribeiro), Нику Роузу (Nick Rose), Сатею Кумару Саху (Satej Kumar Sahu), Михаилу Сизову (Mikhail Sizov), Бену Слатеру (Ben Slater), Пиклсу Спиллу (Pickles Spill), Габриэлю Штутцману (Gabriel Stutzman), Франческо Турко (Francesco Turco), Вольфраму Вольпи (Wolfram Volpi), Боянгу Вану (Boyang Wang), Вальтеру Вежбе (Valter Wierzba) и Кристиану Ветриху (Christian Wuethrich).

В работе над вторым изданием, выпущенным издательством No Starch Press, участвовали: Серена Янг (Serena Yang), заведующая производством, и Мег Снирингер (Meg Sneeringer), выпускающий редактор; Крис Кливленд (Chris Cleveland), редактор; Жорди Гутьеррес Эрмосо (Jordi Gutiérrez Hermoso), научный редактор; и Ким Вимпсетт (Kim Wimpsett), редактор-корректор.

И наконец, спасибо читателям LinuxCommand.org, приславшим мне так много добрых писем. Их поддержка помогла поверить, что я действительно чего-то стою!

Введение

Я хочу поведать вам историю. Нет, не о том, как в 1991-м Линус Торвальдс создал первую версию ядра Linux. Эту историю вы прочитаете в других книгах о Linux. Я не стану рассказывать вам, как несколькими годами ранее Ричард Столлман запустил проект GNU по созданию свободной Unix-подобной операционной системы. И эту апокрифическую историю можно узнать из других книг о Linux.

Но я хочу рассказать, как можно вернуть управление своим компьютером.

В конце 1970-х, когда я, будучи студентом колледжа, только начинал заниматься компьютерами, еще продолжалась IT-революция. Появление микропроцессора сделало возможным приобретение компьютеров простыми людьми, такими как вы или я. Сегодня многим трудно представить себе мир, в котором компьютеры (огромные вычислительные машины) принадлежали только крупным компаниям и правительствам, а обычным людям они были недоступны.

Современный мир сильно изменился. Компьютеры повсюду, от крошечных наручных часов до гигантских вычислительных центров, разбросанных по всему миру. Вдобавок к вездесущим компьютерам у нас появились и сети, связывающие их друг с другом, и благодаря всему этому для нас начался новый век небывалых личных возможностей и творческой свободы. Но обратили ли вы внимание, что за последние два десятилетия стало происходить кое-что еще? Несколько гигантских корпораций захватили контроль над большинством компьютеров в мире и стали решать за вас, что можно и чего нельзя делать с ним. Множество людей по всему миру противостоят этому. Они борются за сохранение контроля над своими компьютерами, создавая собственное программное обеспечение. Они строят Linux.

Сейчас принято употреблять термин «свобода» в отношении Linux, но я не думаю, что большинство знает, что в действительности подразумевается под свободой. Свобода — это возможность самому решать, что будет делать ваш компьютер, и единственный путь к достижению этой свободы — знание того, что он делает.

Свобода — это компьютер без секретов, в котором все можно узнать, если только не лениться.

Зачем нужна командная строка?

Обращали ли вы внимание, что в фильмах, когда «суперхакер» — парень, способный за 30 секунд взломать суперзащищенную военную систему, — садится за компьютер, он никогда не берется за мышь? Создатели фильмов инстинктивно понимают, что мы, будучи людьми, можем сделать за компьютером что-то действительно стоящее, только вводя команды с клавиатуры.

Большинство современных пользователей компьютеров знакомы только с *графическим интерфейсом* (Graphical User Interface, GUI) и верят производителям и экспертам, что *интерфейс командной строки* (Command Line Interface, CLI) — это раннее средневековье. Открою тайну: интерфейс командной строки — удобный и выразительный способ общения с компьютером, во многом напоминающий способ письменного общения между людьми. Как однажды было подмечено, «графический пользовательский интерфейс делает простые задачи еще проще, а интерфейс командной строки делает сложные задачи выполнимыми», — это высказывание остается истинным и по сей день.

Поскольку операционная система Linux создавалась на основе семейства операционных систем Unix, она унаследовала богатое разнообразие инструментов командной строки Unix. ОС Unix заняла ведущее положение в начале 1980-х (хотя появилась на десяток лет раньше), еще до повсеместного распространения графического интерфейса, и, соответственно, широко использовала интерфейс командной строки. Фактически одной из основных причин, по которой первопроходцы Linux выбрали эту ОС, а не, скажем, Windows NT, была мощная поддержка интерфейса командной строки, который «делает сложные задачи выполнимыми».

О чем эта книга

Эта книга представляет обширный обзор «жизни» в командной строке Linux. В отличие от других книг, посвященных одной программе, такой как командный интерпретатор bash, в этой книге я попытаюсь рассказать, как поладить с интерфейсом командной строки в более широком аспекте. Как он работает? Что можно сделать с его помощью? Как лучше его использовать?

Эта книга не об администрировании системы Linux. Даже притом, что любое серьезное обсуждение командной строки неизменно ведет к рассмотрению тем администрирования системы, эта книга затрагивает лишь узкий круг задач, имеющих

отношение к администрированию. Но она готовит читателя к дополнительным исследованиям, закладывая основы знаний, необходимых для использования командной строки как основного инструмента для решения любых серьезных задач системного администрирования.

Эта книга исключительно о Linux. Многие книги пытаются расширить свою целевую аудиторию, включая в содержание другие платформы, такие как Unix и macOS. По этой причине в них рассматриваются лишь общие темы. Эта книга, напротив, посвящена исключительно современным дистрибутивам Linux. И хотя девяносто пять процентов сведений будут полезны пользователями других Unix-подобных систем, основной целевой аудиторией этой книги являются пользователи командной строки современных версий Linux.

Кому адресована эта книга

Эта книга адресована новым пользователям Linux, мигрирующим с других платформ. Весьма вероятно, что вы — «опытный пользователь» определенной версии Microsoft Windows. Возможно, руководитель дал вам задание освоить администрирование Linux-сервера или вы вступаете в новый захватывающий мир одноплатных компьютеров (Single Board Computers, SBC), таких как Raspberry Pi. А может быть, вы обычный пользователь, уставший от нескончаемых проблем безопасности и решивший попробовать Linux. Кем бы вы ни были, здесь вас ждет радушный прием.

Однако следует отметить, что в освоении Linux нет простых путей. Изучение командной строки — непростая задача, требующая определенных усилий. Не то чтобы это чересчур сложно, скорее *очень многообразно*. Обычная система Linux содержит *тысячи* программ, которые можно использовать в командной строке. Поэтому имейте в виду, что желание изучить командную строку должно быть осознанным и целенаправленным.

С другой стороны, изучение командной строки Linux чрезвычайно полезно. Если вы считаете себя «опытным пользователем», подождите немного, и вы узнаете, что такое действительно опытный пользователь. Кроме того, в отличие от других навыков работы с компьютером, умение работать в командной строке еще долго будет оставаться полезным. Навыки, приобретенные сегодня, останутся полезными и через 10 лет. Командная строка выдержала испытание временем.

Если у вас нет опыта программирования — не волнуйтесь, мы поможем в его приобретении.

Что дается в этой книге

В этой книге материал излагается в тщательно выверенной последовательности, как в школе, где учитель руководит вами и направляет вас по правильному пути. Многие авторы грешат тем, что подают материал в «систематическом» порядке, имеющем определенный смысл для писателя, но способном вызывать путаницу у начинающих пользователей.

Цель данной книги — познакомить вас с идеологией Unix, которая отличается от идеологии Windows. По пути мы иногда будем отклоняться в сторону, чтобы попытаться понять, почему то или иное работает именно так, а не иначе. Linux — это не просто программное обеспечение, это также часть обширной культуры Unix, имеющей свой язык и историю. Здесь я мог бы добавить еще пару напыщенных фраз. Но воздержусь от этого.

Книга делится на четыре части, каждая из которых охватывает определенный аспект владения командной строкой:

- **Часть I «Командная оболочка»** вводит в курс основ языка командной строки: структура команд, приемы навигации в файловой системе, редактирование командной строки и поиск справочной информации с описанием команд.
- **Часть II «Окружение и настройка»** посвящена редактированию конфигурационных файлов, управляющих работой командной строки.
- **Часть III «Типичные задачи и основные инструменты»** исследует множество типовых задач, часто выполняемых в командной строке. Unix-подобные операционные системы, такие как Linux, имеют множество «классических» программ командной строки, помогающих выполнять различные операции с данными.
- **Часть IV «Сценарии командной оболочки»** знакомит с программированием на языке командной оболочки, который, по общему мнению, обладает не слишком широкими возможностями, но прост в изучении и позволяет автоматизировать многие вычислительные задачи. Изучая программирование на языке командной оболочки, вы познакомитесь с идеями, которые сможете применять в других языках программирования.

Как читать эту книгу

Начните с начала и последовательно двигайтесь в направлении последней страницы. Это не справочник; книга действительно имеет начало, середину и конец.

ПОЧЕМУ Я НЕ ИСПОЛЬЗУЮ НАЗВАНИЕ «GNU/LINUX»

В некоторых кругах операционную систему Linux принято называть «операционной системой GNU/Linux». Проблема в том, что не существует абсолютно правильного названия, так как эта система создавалась множеством разных людей по всему миру. С технической точки зрения Linux — это название ядра операционной системы, и ничего более. Ядро играет важную роль, конечно, потому что обеспечивает работу операционной системы, но одного его, разумеется, недостаточно.

Ричард Столлман (Richard Stallman), гениальный философ, основатель движения свободного программного обеспечения (Free Software), фонда свободных программ (Free Software Foundation) и проекта GNU, автор первой версии компилятора GNU C Compiler (GCC) и общественной лицензии GNU General Public License (GPL) и прочая и прочая, настаивает на названии «GNU/Linux» как отражающем вклад проекта GNU. Но даже при том, что проект GNU предшествовал появлению ядра Linux и вклад проекта заслуживает самой высокой оценки, использование его названия в названии операционной системы можно расценивать как несправедливость в отношении всех остальных. Кроме того, я считаю, что технически более точным было бы название «Linux/GNU», потому что сначала загружается ядро, а потом все остальное выполняется на его основе.

Под общепринятым названием «Linux» подразумевается ядро и все остальное бесплатное и открытое программное обеспечение, которое можно найти в типичном дистрибутиве Linux, — то есть вся экосистема Linux, а не только компоненты GNU. Кроме того, на рынке операционных систем чаще отдается предпочтение названиям из одного слова, например: DOS, Windows, macOS, Solaris, Irix, AIX. Я решил использовать популярную форму. Но если вы предпочитаете название «GNU/Linux», подставляйте мысленно недостающую часть, когда будете читать эту книгу. Я не буду возражать.

Предварительные условия

Для работы с книгой вам понадобится действующая система Linux. Вы можете получить ее двумя способами:

- **Установить Linux на (пусть и не самый новый) компьютер.** Выбор дистрибутива не играет большой роли, однако многие в наши дни начинают с Ubuntu, Fedora или OpenSUSE. Если не знаете, на чем остановить свой выбор, попробуйте сначала Ubuntu. Установка современного дистрибутива Linux может быть смехотворно простой или чрезвычайно сложной, все зависит от комплектации вашего компьютера. Я бы рекомендовал выбрать не слишком пожилой настольный компьютер, имеющий хотя бы 2 Гбайт ОЗУ и 6 Гбайт свободного дискового пространства. Не советую использовать ноутбуки с беспроводным

подключением к сети, если это возможно, потому что часто они сложнее в настройке.

- **Использовать Live CD.** Одна из самых удобных возможностей, которой обладают многие дистрибутивы Linux, — загрузка и запуск системы непосредственно с компакт-диска или флешки, без необходимости устанавливать ее. Просто включите возможность загрузки с компакт-диска или флешки в настройках BIOS, вставьте Live CD в CD-ROM или флешку в порт USB и перезагрузитесь. Этот способ дает отличную возможность проверить совместимость компьютера с Linux перед установкой. Недостаток Live CD — очень медленная работа в сравнении с установкой Linux на жесткий диск. Оба дистрибутива, Ubuntu и Fedora (среди прочих), имеют версии Live CD.

Вне зависимости от того, как вы установите Linux, чтобы выполнять примеры из книги, вам будут нужны привилегии суперпользователя (то есть администратора).

Получив действующую систему, начинайте знакомиться с материалами книги и выполняйте предлагаемые примеры на своем компьютере. Книга носит сугубо практический характер, поэтому устраивайтесь поудобнее, и вперед!

Что нового во втором издании

Базовая структура и содержание книги остались практически неизменными, но в этом издании «Командной строки Linux» появилось множество новых уточнений, пояснений и улучшений, большинство из которых основано на отзывах читателей. Кроме того, в книгу внесено два особенно заметных улучшения. Во-первых, теперь в книге используется версия `bash` 4.x, которая еще не получила широкого распространения к моменту, когда я работал над первым изданием. В этой четвертой версии `bash` появилось несколько интересных особенностей, описываемых в этом издании. Во-вторых, внесены улучшения в четвертую часть «Сценарии командной оболочки» — теперь в ней представлены более удачные примеры и улучшенные приемы программирования. Сценарии, включенные в четвертую часть, были пересмотрены с целью повышения их надежности и исправления некоторых ошибок ;-).

Ваши отзывы важны для нас!

Эта книга постоянно дорабатывается и обновляется, подобно многим проектам с открытым исходным кодом. Если вы обнаружите техническую ошибку, черкните мне пару строк по адресу *bshotts@users.sourceforge.net*.

Обязательно укажите издание книги, где вы нашли ошибку. Ваши исправления и предложения вполне могут попасть в будущие издания.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Часть I

КОМАНДНАЯ ОБОЛОЧКА

1 Что такое командная оболочка

Говоря о командной строке, на самом деле мы имеем в виду *командную оболочку* (*shell*). Командная оболочка — это программа, которая принимает команды, введенные с клавиатуры, и передает их операционной системе для выполнения. Практически все дистрибутивы Linux поставляются с командной оболочкой из проекта GNU, которая называется `bash`. Имя `bash` — это аббревиатура от названия *B*ourne *Again Sh*ell, отражающего тот факт, что `bash` является улучшенной заменой `sh`, первоначальной командной оболочки для Unix, написанной Стивом Борном (Steve Bourne).

Эмуляторы терминалов

При использовании графического интерфейса для взаимодействия с командной оболочкой нам понадобится еще одна программа — *эмулятор терминала*. Заглянув в меню рабочего стола, вы наверняка обнаружите такую программу. В KDE используется konsole, в GNOME — gnome-terminal, однако соответствующий пункт в меню часто называется просто «Terminal» (или «Терминал»). Для Linux существует также множество других эмуляторов терминала, но все они решают одну и ту же задачу: предоставляют доступ к командной оболочке. Со временем у вас наверняка появятся свои предпочтения, в зависимости от «рюшечек и бантиков», которые они имеют.

Первые удары по клавишам

Итак, приступим. Запустите эмулятор терминала. После появления окна на экране вы увидите в нем нечто подобное:

```
[me@linuxbox ~]$
```

НЕСКОЛЬКО СЛОВ О МЫШИ И ФОКУСЕ ВВОДА

Для работы с командной оболочкой достаточно одной клавиатуры, однако эмулятор терминала позволяет также использовать мышь. X Window System (механизм, который воспроизводит графический интерфейс на экране) поддерживает прием быстрого копирования с помощью мыши. Если выделить текст, нажав левую кнопку и переместив указатель мыши над ним (или выполнив двойной щелчок на слове), он будет скопирован в специальный буфер, которым управляет X. Нажатие средней кнопки мыши вызовет вставку текста в позицию курсора. Попробуйте проделать этот фокус.

Не пытайтесь использовать комбинации CTRL+C и CTRL+V для копирования и вставки в окне терминала. Эти команды там не работают. В командной оболочке эти комбинации клавиш имеют другие значения, присвоенные им задолго до появления Microsoft Windows.

Графическое окружение вашего рабочего стола (скорее всего, KDE или GNOME) работает очень похоже на графическое окружение Windows и, вероятнее всего, реализует политику «щелкни, чтобы передать фокус ввода». Это означает, что для передачи фокуса ввода в окно (его активизации) на нем нужно щелкнуть мышью. Это противоречит традиционному поведению X «фокус следует за мышью», когда для передачи фокуса ввода в окно достаточно просто навести на него указатель мыши. Окно не поднимется на передний план, пока вы не щелкнете на нем мышью, но способно принять фокус ввода. Настройка политики «фокус следует за мышью» упростит работу с окном терминала. Попробуйте, я думаю, вам понравится. Соответствующие параметры находятся в программе настройки вашего диспетчера окон.

Это называется *приглашением к вводу* (*shell prompt*) и появляется всякий раз, когда командная оболочка готова принять ввод. В разных дистрибутивах приглашение выглядит по-разному, но обычно включает строку *имя_пользователя@имя_компьютера*, за которой следует имя текущего каталога (подробнее об этом чуть ниже) и знак доллара.

Если последний символ в приглашении — знак решетки (#), а не знак доллара, это означает, что сеанс в терминале обладает привилегиями суперпользователя. То есть либо вы зарегистрировались как пользователь root, либо запустили эмулятор терминала, который автоматически устанавливает привилегии суперпользователя (администратора).

Будем считать, что пока все идет хорошо, и попробуем что-нибудь ввести. Наберите на клавиатуре какую-нибудь бессмыслицу, например:

```
[me@linuxbox ~]$ kaekfjaeifj
```

Поскольку это бессмыслица, командная оболочка немедленно сообщит об этом и даст вам второй шанс:

```
bash: kaekfjaeifj: команда не найдена
[me@linuxbox ~]$
```

История команд

Если теперь нажать клавишу со стрелкой вверх, после приглашения к вводу появится предыдущая команда `kaekfjaeifj`. Это называется *историей команд*. Большинство дистрибутивов Linux по умолчанию запоминают последние 1000 команд. Нажмите клавишу со стрелкой вниз, и предыдущая команда исчезнет.

Управление курсором

Вызовите предыдущую команду, еще раз нажав клавишу со стрелкой вверх. Теперь попробуйте понажимать клавиши со стрелками влево и вправо. Видите, как меняется позиция курсора в командной строке? Благодаря этому легко можно редактировать команды.

Некоторые простые команды[1]

Теперь, когда вы понажимали клавиши, попробуем ввести несколько простых команд. Первая команда `date`. Она выводит текущие время и дату:

```
[me@linuxbox ~]$ date
Fri Feb 2 15:09:41 EST 2018
```

Родственная ей команда `cal` по умолчанию выводит календарь текущего месяца:

```
[me@linuxbox ~]$ cal
    February 2018
Su Mo Tu We Th Fr Sa
             1  2  3
```

[1] Часть вывода команд в Linux переведена на русский язык, часть — нет. В дальнейшем те системные сообщения и результаты выполнения команд, выводимые на консоль, которые в локализованной версии Linux переведены, мы будем приводить на русском, остальные — на английском. В тех случаях, когда русский перевод консольного вывода важен для изложения, он будет приводиться в сносках внизу страницы. — *Примеч. ред.*

```
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28
```

Чтобы увидеть объем свободного пространства на дисках, введите `df`:

```
[me@linuxbox ~]$ df
Файл.система      1K-блоков   Использовано  Доступно  Использовано%  Смонтировано в
/dev/sda2          15115452        5012392   9949716            34%  /
/dev/sda5          59631908       26545424  30008432            47%  /home
/dev/sda1            147764          17370    122765            13%  /boot
tmpfs               256856               0    256856             0%  /dev/shm
```

Аналогично, чтобы увидеть объем свободного пространства в памяти, введите команду `free`:

```
[me@linuxbox ~]$ free
              total        used        free      shared  buffers   available
Mem:         513712      503976        9736           0     5312      122916
-/+ buffers/cache:       375748      137964
Swap:       1052248      104712      947536
```

Завершение сеанса работы с терминалом

Завершить сеанс работы с терминалом можно, либо закрыв окно эмулятора терминала, либо введя команду `exit`:

```
[me@linuxbox ~]$ exit
```

КОНСОЛЬ ЗА КУЛИСАМИ

Даже если не запущен ни один эмулятор терминала, за ширмой графического рабочего стола продолжают выполняться несколько сеансов терминалов. Получить доступ к этим сеансам, или *виртуальным консолям*, в большинстве дистрибутивов Linux можно с помощью комбинаций клавиш, начиная с CTRL+ALT+F1 до CTRL+ALT+F6. После перехода к сеансу вы увидите приглашение к регистрации в системе, где нужно ввести имя пользователя и пароль. Для переключения из одной виртуальной консоли в другую используйте клавиши ALT и F1–F6. Чтобы вернуться в графическое окружение рабочего стола, в большинстве систем нужно нажать ALT+F7.

Заключение

Эта глава ознаменовала начало нашего путешествия по командной строке Linux введением в командную оболочку, показав, как запустить и завершить сеанс работы с терминалом. Мы также увидели работу нескольких простых команд и узнали, как выполняются простейшие операции редактирования в командной строке. Это было совсем не страшно, правда?

В следующей главе мы познакомимся еще с несколькими командами и побродим по файловой системе Linux.

2 Навигация

Первое, что мы попробуем изучить (после пробных нажатий на клавиши), — навигацию в файловой системе Linux. В этой главе введем в обиход следующие команды:

`pwd` — выводит название текущего рабочего каталога;

`cd` — выполняет переход в другой каталог;

`ls` — выводит список содержимого каталога.

Дерево каталогов файловой системы

Так же как Windows, Unix-подобная операционная система, такая как Linux, организует свои файлы в *иерархическую структуру каталогов*. То есть каталоги (в других системах их иногда называют *папками*) имеют древовидную организацию и могут содержать файлы и другие каталоги. Первый каталог в файловой системе называется *корневым каталогом*. Корневой каталог содержит файлы и подкаталоги, которые в свою очередь также содержат файлы и каталоги, и т. д.

Обратите внимание, что в отличие от Windows, где для каждого устройства хранения создается отдельная файловая система, в Unix-подобных системах, таких как Linux, всегда имеется только одна файловая система, независимо от числа приводов или устройств хранения, подключенных к компьютеру. Устройства хранения подключаются (или, как принято говорить, *монтируются*) к разным точкам дерева в соответствии с желанием системного администратора, человека (или нескольких человек), ответственного за обслуживание системы.

Текущий рабочий каталог

Многие из нас наверняка знакомы с графическими диспетчерами файлов, представляющими дерево каталогов файловой системы, как показано на рис. 2.1.

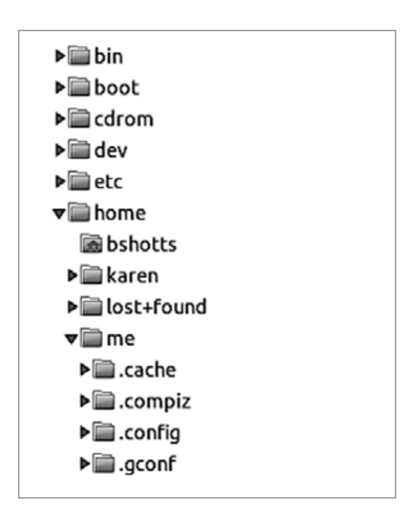


Рис. 2.1. Дерево каталогов файловой системы в диспетчере файлов с графическим интерфейсом

Обратите внимание, что обычно дерево отображается в перевернутом виде, то есть с корнем наверху и ветвями, направленными вниз.

Однако командная строка не имеет графического интерфейса, поэтому для перемещения по дереву файловой системы его следует представлять иначе.

Представьте файловую систему в виде лабиринта в форме перевернутого дерева и себя в середине. В любой конкретный момент времени мы можем находиться только в одном каталоге, видеть файлы в этом каталоге и перемещаться в вышележащий каталог (называется *родительским каталогом*) и в любые нижележащие каталоги. Каталог, в котором мы находимся, называется *текущим рабочим каталогом*. Название текущего рабочего каталога выводится командой `pwd` (print working directory — вывести рабочий каталог):

```
[me@linuxbox ~]$ pwd
/home/me
```

Сразу после входа в систему (или запуска сеанса в эмуляторе терминала) текущим рабочим каталогом становится наш *домашний каталог*. Каждый пользователь

имеет свой домашний каталог, который является единственным, где пользователю позволено осуществлять запись в файлы, когда он действует с привилегиями обычного пользователя.

Вывод содержимого каталога

Чтобы вывести список файлов и каталогов в текущем рабочем каталоге, воспользуйтесь командой `ls`:

```
[me@linuxbox ~]$ ls
Desktop  Documents  Music  Pictures  Public  Templates  Videos
```

В действительности командой `ls` можно вывести содержимое любого, не только текущего, рабочего каталога, а также получить массу дополнительной любопытной информации, но об этом мы поговорим в главе 3.

Смена текущего рабочего каталога

Чтобы сменить рабочий каталог (в котором мы находимся в середине древовидного лабиринта), можно воспользоваться командой `cd`: введите `cd` и добавьте путь к желаемому рабочему каталогу. Путь (pathname) — это маршрут, перечисляющий ветви дерева, по которым нужно пройти, чтобы достигнуть желаемого каталога. Пути могут определяться двумя способами: как *абсолютные* или как *относительные*. Рассмотрим сначала абсолютные пути.

Абсолютные пути

Абсолютный путь начинается с корневого каталога и перечисляет ветви дерева, отделяющие корень от желаемого каталога или файла. Например, в системе имеется каталог, в который устанавливается большинство программ. Путь к этому каталогу имеет вид */usr/bin*. То есть в корневом каталоге (представлен первым символом «слеш» в пути) имеется каталог с названием *usr*, содержащий каталог с названием *bin*.

```
[me@linuxbox ~]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
[me@linuxbox bin]$ ls
...Длинный, очень длинный список файлов...
```

Как видите, мы сменили текущий рабочий каталог на */usr/bin*, и он полон файлов. Обратите внимание, как изменилось приглашение командной оболочки к вводу. Для удобства оно обычно настраивается так, чтобы автоматически показывать название рабочего каталога.

Относительные пути

В отличие от абсолютного пути, начинающегося в корневом каталоге и ведущего к каталогу назначения, относительный путь начинается в рабочем каталоге. Для обозначения относительных позиций в дереве файловой системы используется пара специальных символов: `.` (точка) и `..` (точка-точка).

Символ `.` (точка) обозначает рабочий каталог, а символ `..` (точка-точка) обозначает каталог, родительский по отношению к рабочему. Ниже показано, как ими пользоваться. Давайте снова сменим рабочий каталог на */usr/bin*:

```
[me@linuxbox ~]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
```

Отлично, а теперь допустим, что мы хотим сменить рабочий каталог на родительский для каталога */usr/bin*, которым является */usr*. Сделать это можно двумя способами: пойти по абсолютному пути:

```
[me@linuxbox bin]$ cd /usr
[me@linuxbox usr]$ pwd
/usr
```

или по относительному:

```
[me@linuxbox bin]$ cd ..
[me@linuxbox usr]$ pwd
/usr
```

Два разных способа дают идентичные результаты. И каким же лучше пользоваться? Конечно, тем, который требует нажимать меньше клавиш!

Аналогично, существуют два способа сменить рабочий каталог с */usr* на */usr/bin*. Указать абсолютный путь:

```
[me@linuxbox usr]$ cd /usr/bin
[me@linuxbox bin]$ pwd
/usr/bin
```

или относительный:

```
[me@linuxbox usr]$ cd ./bin
[me@linuxbox bin]$ pwd
/usr/bin
```

А теперь я хочу сделать важное замечание. Практически во всех случаях можно опустить пару символов `./`, потому что они подразумеваются по умолчанию. Команда

```
[me@linuxbox usr]$ cd bin
```

даст тот же результат. Вообще если путь к чему-либо не указан явно, подразумевается текущий рабочий каталог.

ЧТО СЛЕДУЕТ ЗНАТЬ ОБ ИМЕНАХ ФАЙЛОВ

В системе Linux файлы именуются так же, как в других системах, таких как Windows, но есть некоторые важные отличия.

- Файлы, имена которых начинаются с точки, считаются скрытыми. Это означает, что команда `ls` не будет выводить их, если не вызвать ее с параметром: `ls -a`. В момент создания учетной записи пользователя в его домашний каталог помещается несколько скрытых файлов, где хранятся различные настройки учетной записи. В главе 11 мы еще вернемся к подобным файлам, когда будем рассматривать возможные настройки своего окружения. Кроме того, некоторые приложения помещают в домашний каталог свои скрытые файлы с настройками.
- Linux, как это принято в Unix, различает регистр символов в именах файлов и командах. Файлы с именами *File1* и *file1* — это разные файлы.
- Хотя Linux поддерживает длинные имена файлов с пробелами и знаками пунктуации, старайтесь не использовать в именах файлов другие знаки пунктуации, кроме точки, дефиса и подчеркивания. *И самое главное, не используйте пробелы в именах файлов*. Если необходимо отделить друг от друга слова в имени файла, используйте символы подчеркивания. Потом вы не раз скажете себе спасибо за это.
- В Linux не поддерживается понятие «расширения файла», как в некоторых других операционных системах. Вы можете давать своим файлам любые имена. Тип и/или назначение файла определяется другими средствами. Но даже при том, что Unix-подобные операционные системы не используют расширения файлов для определения типа/назначения файлов, некоторые прикладные программы все же используют их для этой цели.

Некоторые полезные сокращения

В табл. 2.1 перечислены некоторые способы быстрой смены рабочего каталога.

Таблица 2.1. Сокращенные варианты команды cd

Сокращение	Результат
`cd`	Сменить рабочий каталог на домашний
`cd -`	Сменить рабочий каталог на предыдущий рабочий каталог
`cd ~username`	Сменить рабочий каталог на домашний каталог пользователя *username*. Например, `cd ~bob` выполнит переход в домашний каталог пользователя `bob`

Заключение

Эта глава рассказала, как командная оболочка представляет структуру каталогов системе. Мы узнали об абсолютных и относительных путях и познакомились с основными командами для перемещения по этой структуре. В следующей главе мы используем эти знания в процессе знакомства с современной системой Linux.

3 Исследование системы

Теперь, когда мы знаем, как перемещаться по файловой системе, совершим обзорное путешествие по системе Linux. Но прежде чем отправиться, познакомимся еще с несколькими командами, которые пригодятся в пути:

`ls` — выводит список содержимого каталога;

`file` — определяет тип файла;

`less` — выводит содержимое файла.

Любопытные возможности ls

Команда `ls` является, пожалуй, одной из самых часто используемых команд, и не без оснований. С ее помощью можно увидеть, что находится в каталоге, и узнать некоторые важные атрибуты файлов и каталогов. Как мы уже видели, чтобы получить список файлов и подкаталогов в текущем рабочем каталоге, достаточно ввести команду `ls`:

```
[me@linuxbox ~]$ ls
Desktop  Documents  Music  Pictures  Public  Templates  Videos
```

Команде можно явно указать каталог, содержимое которого требуется вывести:

```
me@linuxbox ~]$ ls /usr
bin  games  include  lib  local  sbin  share  src
```

и даже несколько каталогов. Следующая команда выведет содержимое домашнего каталога пользователя (обозначен символом ~) и каталога */usr*:

```
[me@linuxbox ~]$ ls ~ /usr
/home/me:
Desktop  Documents  Music  Pictures  Public  Templates  Videos
/usr:
bin  games  include  lib  local  sbin  share  src
```

Можно также изменить формат вывода, чтобы получить больше информации:

```
[me@linuxbox ~]$ ls -l
total 56
drwxrwxr-x 2 me me 4096 2017-10-26 17:20 Desktop
drwxrwxr-x 2 me me 4096 2017-10-26 17:20 Documents
drwxrwxr-x 2 me me 4096 2017-10-26 17:20 Music
drwxrwxr-x 2 me me 4096 2017-10-26 17:20 Pictures
drwxrwxr-x 2 me me 4096 2017-10-26 17:20 Public
drwxrwxr-x 2 me me 4096 2017-10-26 17:20 Templates
drwxrwxr-x 2 me me 4096 2017-10-26 17:20 Videos
```

Параметр `-l`, добавленный в команду, требует использования «длинного» (long) формата вывода.

Параметры и аргументы

Мы подошли к очень важному моменту, касающемуся особенностей работы большинства команд. Команды часто сопровождаются одним или несколькими *параметрами*, изменяющими их поведение, и дополнительными, одним или несколькими, *аргументами*, на которые воздействует команда. Поэтому большинство команд выглядят примерно так:

```
команда -параметры аргументы
```

Большинство команд используют параметры, состоящие из одного символа, которому предшествует дефис, например: `-l`. Но многие команды, в том числе команды из проекта GNU, поддерживают *параметры с длинными именами*, состоящие из слова, которому предшествуют два дефиса. Кроме того, многие команды позволяют объединять вместе параметры с короткими именами. В следующем примере команде `ls` передаются два параметра: параметр `l`, требующий использовать длинный (long) формат вывода, и параметр `t`, требующий сортировать результаты по времени (time) изменения:

```
[me@linuxbox ~]$ ls -lt
```

Добавим параметр с длинным именем `--reverse`, чтобы изменить порядок сортировки на обратный:

```
[me@linuxbox ~]$ ls -lt --reverse
```

ПРИМЕЧАНИЕ

Параметры команд, подобно именам файлов в Linux, чувствительны к регистру символов.

Команда `ls` имеет огромное число допустимых параметров. Наиболее популярные из них перечислены в табл. 3.1.

Таблица 3.1. Наиболее популярные параметры команды ls

Параметр	Длинный параметр	Описание
`-a`	`--all`	Список всех (all) файлов, даже с именами, начинающимися с точки, которые обычно не выводятся (то есть скрытых)
`-A`	`--almost-all`	Действует подобно параметру `-a`, но не выводит каталоги `.` (текущий рабочий каталог) и `..` (родительский каталог)
`-d`	`--directory`	Обычно в присутствии этого параметра команда `ls` выводит информацию о самом каталоге, а не его содержимое. Используйте этот параметр в сочетании с параметром `-l`, чтобы получить дополнительную информацию о каталоге, а не о его содержимом
`-F`	`--classify`	Добавляет в конец каждого имени символ-индикатор (например, прямой слеш, если это имя каталога)
`-h`	`--human-readable`	При использовании длинного формата вывода отображает размеры файлов не в байтах, а в величинах с единицами измерения
`-l`		Выводит результаты с использованием длинного формата
`-r`	`--reverse`	Выводит результаты в обратном порядке. Обычно команда `ls` выводит результаты в алфавитном порядке
`-S`		Сортировать результаты по размеру (size)
`-t`		Сортировать результаты по времени (time) последнего изменения

Пристальный взгляд на длинный формат

Как было показано выше, параметр `-l` заставляет команду `ls` выводить результаты с использованием длинного формата. Этот формат предусматривает вывод

большого количества полезной информации. Ниже приводится пример вывода содержимого каталога `Examples` в системе Ubuntu:

```
-rw-r--r-- 1 root root 3576296 2017-04-03 11:05 Experience ubuntu.ogg
-rw-r--r-- 1 root root 1186219 2017-04-03 11:05 kubuntu-leaflet.png
-rw-r--r-- 1 root root   47584 2017-04-03 11:05 logo-Edubuntu.png
-rw-r--r-- 1 root root   44355 2017-04-03 11:05 logo-Kubuntu.png
-rw-r--r-- 1 root root   34391 2017-04-03 11:05 logo-Ubuntu.png
-rw-r--r-- 1 root root   32059 2017-04-03 11:05 oo-cd-cover.odf
-rw-r--r-- 1 root root  159744 2017-04-03 11:05 oo-derivatives.doc
-rw-r--r-- 1 root root   27837 2017-04-03 11:05 oo-maxwell.odt
-rw-r--r-- 1 root root   98816 2017-04-03 11:05 oo-trig.xls
-rw-r--r-- 1 root root  453764 2017-04-03 11:05 oo-welcome.odt
-rw-r--r-- 1 root root  358374 2017-04-03 11:05 ubuntu Sax.ogg
```

Рассмотрим различные поля для одного из файлов и их назначение (табл. 3.2).

Таблица 3.2. Поля длинного формата вывода команды ls

Поле	Назначение
`-rw-r-r--`	Права доступа к файлу. Первый символ указывает тип файла. Например, символом дефиса обозначаются обычные файлы, а символом `d` — каталоги. Следующие три символа сообщают о правах доступа для владельца файла, следующие три — для членов группы, которой принадлежит файл, и последние три — для всех остальных. Более полный обзор прав доступа приводится в главе 9
`1`	Число жестких ссылок на файл. Подробнее о ссылках рассказывается в разделах «Символические ссылки» и «Жесткие ссылки» далее в этой главе
`root`	Имя пользователя, владеющего файлом
`root`	Имя группы, владеющей файлом
`32059`	Размер файла в байтах
`2017-04-03 11:05`	Дата и время последнего изменения файла
`oo-cd-cover.odf`	Имя файла

Определение типов файлов командой file

Занимаясь исследованием системы, полезно иметь возможность определять тип содержимого файлов. В этом нам поможет команда `file`. Как отмечалось выше,

имена файлов в Linux не обязаны отражать тип содержимого файлов. Например, увидев имя файла *picture.jpg*, можно предположить, что он содержит изображение в формате JPEG, но в Linux такие предположения могут не оправдываться. Вызвать команду `file` можно так:

```
file имя_файла
```

Команда `file` выводит краткое описание содержимого файла. Например:

```
[me@linuxbox ~]$ file picture.jpg
picture.jpg: JPEG image data, JFIF standard 1.01
```

Существует множество разных типов файлов. Одна из известных идей в Unix-подобных системах, таких как Linux, гласит: «Все сущее есть файл». По мере чтения книги вы убедитесь в истинности этого утверждения.

Типы многих файлов в вашей системе будут вам знакомы, например файлы MP3 и JPEG, но иногда будут попадаться файлы с малоизвестными и даже странными типами.

Просмотр содержимого файлов командой less

Команда `less` — это программа для просмотра текстовых файлов. В системе Linux присутствует множество файлов, содержащих обычный удобочитаемый текст. Программа `less` предоставляет удобный способ исследовать их содержимое.

Зачем может понадобиться исследовать текстовые файлы? Дело в том, что многие файлы с системными настройками (их называют *конфигурационными файлами*) хранят информацию в этом формате, что дает возможность прочитать их и вникнуть в особенности работы системы. Кроме того, в этом формате хранятся многие программы в системе (их называют *сценариями*). В последующих главах мы узнаем, как редактировать файлы с настройками системы и как писать свои сценарии, а пока просто просматривайте их содержимое.

Команда `less` используется так:

```
less имя_файла
```

После запуска программа `less` позволяет прокручивать текстовый файл взад и вперед. Например, просмотреть содержимое файла со всеми известными системе учетными записями пользователей можно с помощью следующей команды:

```
[me@linuxbox ~]$ less /etc/passwd
```

> **ЧТО ЕСТЬ «ТЕКСТ»**
>
> Существует множество способов представления информации в компьютере. Все способы связаны с определением отношения между смысловой информацией и числами, которые применяются для ее представления. В конце концов, компьютеры могут работать только с числами, и все данные в компьютере преобразуются в числовое представление.
>
> Некоторые из этих систем представления очень сложны (например, сжатые видеофайлы), другие, напротив, очень просты. Одна из самых ранних и простых систем называется ASCII-текст. ASCII (произносится «ас-ки») — это аббревиатура названия «American Standard Code for Information Interchange» (американский стандартный код для обмена информацией). Эта простая система кодирования впервые была использована в телетайпах.
>
> Текст — это простое отображение «один в один» символов в числа. Это очень компактный формат. Пятьдесят символов текста преобразуются в пятьдесят байт данных. Но это не то же самое, что текст в документе, созданном текстовым процессором, таким как Microsoft Word или LibreOffice Writer. Файлы документов, в отличие от простых файлов с ASCII-текстом, содержат множество нетекстовых элементов, используемых для описания их структуры и форматирования. Файлы с простым ASCII-текстом содержат только сами символы и очень небольшое количество простейших кодов управления, таких как символы табуляции, возврата каретки и перевода строки.
>
> В системе Linux многие файлы хранятся в текстовом формате, и многие инструменты работают с текстовыми файлами. Даже Windows признает важность этого формата. Хорошо известная программа Notepad (Блокнот) — это редактор для простых файлов с ASCII-текстом.

После запуска программа `less` выведет содержимое файла. Если файл занимает больше одной страницы, его можно прокручивать вверх и вниз. Чтобы выйти из программы `less`, нажмите клавишу Q.

В табл. 3.3 перечислены клавиатурные команды, наиболее часто используемые в программе `less`.

Таблица 3.3. Команды программы less

Команда	Действие
Page Up или b	Прокрутка к началу на одну страницу
Page Down или ПРОБЕЛ	Прокрутка к концу на одну страницу

Команда	Действие
СТРЕЛКА ВВЕРХ	Прокрутка к началу на одну строку
СТРЕЛКА ВНИЗ	Прокрутка к концу на одну строку
G	Переход в конец текстового файла
1G или g	Переход в начало текстового файла
/символы	Поиск вниз по тексту до ближайшего вхождения указанной последовательности символов
n	Поиск следующего вхождения искомой последовательности символов
h	Вывод экрана со справкой
q	Завершить `less`

МЕНЬШЕ ЗНАЧИТ БОЛЬШЕ

Программа `less` создавалась как улучшенная замена более ранней Unix-программы с именем *more*. Ее имя — это игра слов «less is more» (меньше значит больше) — девиз архитекторов-модернистов и проектировщиков.

`less` относится к категории программ постраничного просмотра текстовых документов, которые называют *пейджерами* (pagers). В отличие от программы `more`, которая может листать страницы только вперед, программа `less` способна листать текст в обоих направлениях, вперед и назад, и имеет множество других особенностей.

Обзорное путешествие

Файловая система в Linux имеет практически ту же компоновку, что и в других Unix-подобных системах. Фактически ее структура определяется опубликованным стандартом с названием «Linux Filesystem Hierarchy Standard». Не все дистрибутивы Linux следуют этому стандарту, но большинство придерживаются его достаточно близко.

А теперь немного попутешествуем по файловой системе и познакомимся с основными достопримечательностями системы Linux. Это даст нам шанс попрактиковаться в навигации. Первое, что мы обнаружим: многие интересные файлы имеют простой текстовый формат. В ходе путешествия пробуйте выполнить следующие действия:

- с помощью команды `cd` перейдите в указанный каталог;
- выведите содержимое каталога командой `ls -l`;
- если увидите заинтересовавший вас файл, определите его тип командой `file`;
- если файл выглядит как текстовый, попробуйте просмотреть его командой `less`.

> **ВСПОМНИТЕ ТРЮК С КОПИРОВАНИЕМ И ВСТАВКОЙ!**
>
> Если вы пользуетесь мышью, выполните двойной щелчок на имени файла, чтобы скопировать его, и щелчок средней кнопкой, чтобы вставить в команду.

Если по ошибке вы попробовали просмотреть содержимое нетекстового файла и полностью нарушили нормальную работу окна терминала, вы можете вернуть его в исходное состояние, выполнив команду `reset`.

В ходе путешествия не бойтесь заглядывать внутрь системы. Обычные пользователи практически ничего не смогут испортить. Это работа системного администратора! Если команда пожалуется на что-то, просто перейдите к чему-нибудь другому. Потратьте некоторое время на знакомство с окрестностями. Это наша система, и мы вправе заниматься ее исследованием. Помните, что в Linux нет секретов!

В табл. 3.4 перечислены несколько каталогов для исследования. Но вы можете заняться исследованием любых других каталогов! Не бойтесь заглядывать внутрь и экспериментировать!

Таблица 3.4. Каталоги в системе Linux

Каталог	Описание
/	Корневой каталог, откуда все начинается
/bin	Содержит двоичные (binaries) файлы (программы), необходимые для загрузки и функционирования системы
/boot	Содержит ядро Linux, образ начального RAM-диска (с драйверами, необходимыми на этапе загрузки) и сам загрузчик. Интересные файлы: • */boot/grub/grub.conf* или *menu.lst*, используются для настройки загрузчика • */boot/vmlinuz* (или с похожим именем), ядро Linux

Каталог	Описание
/dev	Специальный каталог, содержащий *узлы устройств*. «Все сущее есть файл» применяется также к устройствам. Здесь ядро хранит список всех известных ему устройств
/etc	Каталог */etc* содержит все системные конфигурационные файлы. Здесь же хранится коллекция сценариев командной оболочки, запускающих системные службы во время загрузки. Практически все файлы в этом каталоге содержат обычный читаемый текст. В */etc* все файлы интересны, но, на мой взгляд, особенный интерес представляют: • */etc/crontab*, файл, определяющий время запуска автоматизированных заданий; • */etc/fstab*, таблица устройств хранения и соответствующих им точек монтирования; • */etc/passwd*, список всех учетных записей пользователей
/home	В обычных конфигурациях каждому пользователю выделяется свой домашний каталог в каталоге */home*. Простые пользователи могут записывать что-нибудь только в файлы, находящиеся в их домашних каталогах. Это ограничение защищает систему от необдуманных действий пользователей
/lib	Содержит файлы разделяемых библиотек, используемых основными системными программами. Они напоминают библиотеки DLL в Windows
/lost+found	Каждый раздел или устройство, отформатированные с использованием файловой системы Linux, такой как ext3, будут иметь этот каталог. Он используется на случай частичного восстановления повреждений в файловой системе. Если с системой ничего страшного не происходило, этот каталог будет оставаться пустым
/media	В современных системах Linux каталог */media* будет содержать точки монтирования съемных носителей, таких как USB-диски, CD-ROM и т. д., которые монтируются в момент подключения
/mnt	В старых системах Linux каталог */mnt* содержал точки монтирования съемных носителей, монтируемых вручную
/opt	Каталог */opt* используется для установки «необязательного» (optional) программного обеспечения, в основном для установки коммерческого программного обеспечения
/proc	Специальный каталог. Не является фактической файловой системой, в том смысле, что файлы в этом каталоге не хранятся на жестком диске. Это виртуальная файловая система, поддерживаемая ядром Linux. Файлы в ней являются «глазками», через которые можно заглянуть в ядро. Эти файлы доступны для чтения и помогают «увидеть» компьютер глазами ядра

Таблица 3.4 (*окончание*)

Каталог	Описание
/root	Домашний каталог пользователя root
/sbin	Каталог содержит системные двоичные файлы (system binaries). Эти программы выполняют жизненно важные задачи и обычно запускаются только суперпользователем
/tmp	Каталог */tmp* играет роль временного хранилища для временных файлов, создаваемых разными программами. В некоторых конфигурациях этот каталог принудительно очищается при каждой перезагрузке системы
/usr	Дерево каталогов */usr* является, пожалуй, самым объемным в системе Linux. В нем хранятся все программы и файлы поддержки, используемые обычными пользователями
/usr/bin	Каталог */usr/bin* содержит выполняемые программы, установленные дистрибутивом Linux. Очень часто в этом каталоге хранятся тысячи программ
/usr/lib	Содержит разделяемые библиотеки для программ в */usr/bin*
/usr/local	Дерево каталогов */usr/local* используется для установки тех программ, которые не входят в состав дистрибутива, но должны быть доступны всем пользователям в системе. Программы, собираемые из исходных текстов, обычно устанавливаются в */usr/local/bin*. В новейших версиях системы Linux это дерево каталогов присутствует, но остается пустым, пока системный администратор не добавит туда что-нибудь
/usr/sbin	Содержит дополнительные программы для администрирования
/usr/share	Каталог */usr/share* содержит все разделяемые данные, используемые программами в */usr/bin*, в том числе конфигурационные файлы с настройками по умолчанию, ярлыки, фоновые изображения для рабочего стола, звуковые файлы и т. д
/usr/share/doc	Большинство пакетов, установленных в системе, содержат документацию. Вся эта документация, организованная по пакетам, хранится в каталоге */usr/share/doc*
/var	За исключением */tmp* и */home*, все упоминавшиеся выше каталоги остаются относительно статичными; то есть их содержимое почти не меняется. Дерево каталогов */var* — как раз то место, где хранятся часто изменяемые данные: различные базы данных, буферные файлы, почта пользователей и пр.
/var/log	Каталог */var/log* содержит файлы журналов с записями о различных действиях, выполнявшихся в системе. Они очень важны и должны проверяться время от времени. Наиболее полезными являются файлы */var/log/messages* и */var/log/syslog*. Обратите внимание, что из соображений безопасности некоторые системы требуют привилегий суперпользователя для просмотра файлов журналов

Символические ссылки

Просматривая содержимое каталогов (например, `/lib`), нередко можно увидеть такие записи:

```
lrwxrwxrwx 1 root root 11 2018-08-11 07:34 libc.so.6 -> libc-2.6.so
```

Обратили внимание на первую букву `l` и на присутствие двух имен файлов в конце? Это специальный файл, который называется *символической ссылкой* (иногда их называют *мягкими ссылками* или, на жаргоне, *симлинками*). В большинстве Unix-подобных систем можно дать одному и тому же файлу несколько имен. Даже притом, что на данный момент ценность такого приема может быть не очевидна, в действительности это очень удобная возможность.

Вообразите следующий сценарий: программе требуется некий разделяемый ресурс (например, библиотека), хранящийся в файле с именем «foo», но номер версии «foo» меняется очень часто. Было бы хорошо включить номер версии в имя файла, чтобы администратор или другое заинтересованное лицо могли видеть, какая версия «foo» установлена. И здесь возникает проблема. Если изменить имя разделяемого ресурса, нам придется проверять каждую программу, использующую этот ресурс, и изменять в ней имя ресурса на новое после установки новой версии ресурса. Если честно, такая перспектива не выглядит привлекательной.

Символические ссылки помогут спасти положение. Допустим, мы установили «foo» версии 2.6 с именем файла «foo-2.6» и затем создали символическую ссылку с простым именем «foo», указывающую на ресурс «foo-2.6». То есть когда программа откроет файл «foo», в действительности она откроет файл «foo-2.6». И все будут счастливы. Программы, полагающиеся на имя «foo», найдут нужный файл, а мы сможем увидеть фактическую версию ресурса. Когда придет время обновить ресурс до версии «foo-2.7», мы просто добавим файл в систему, удалим символическую ссылку «foo» и создадим новую символическую ссылку, указывающую на новую версию. Такой подход не только решает проблему обновления версий, но также позволяет сохранить на компьютере обе версии ресурса. Представьте, что в версии «foo-2.7» обнаружилась ошибка (ох уж эти разработчики!) и нужно вернуть старую версию. В этом случае достаточно просто вновь удалить символическую ссылку, указывающую на новую версию, и создать новую символическую ссылку, указывающую на старую версию.

Запись выше (получена в каталоге `/lib` в системе Fedora) соответствует символической ссылке с именем `libc.so.6`, указывающей на файл разделяемой библиотеки с именем *libc-2.6.so*. Это означает, что программа, ищущая *libc.so.6*, фактически получит файл *libc-2.6.so*. Как создавать символические ссылки, мы узнаем в следующей главе.

Жесткие ссылки

Пока мы не ушли далеко от темы ссылок, нужно упомянуть, что существует второй тип ссылок, которые называют *жесткими ссылками* (hard link). Жесткие ссылки также позволяют присвоить одному файлу несколько имен, но они действуют иначе. Подробнее о различиях между жесткими и символическими ссылками рассказывается в следующей главе.

Заключение

Совершив обзорное путешествие, мы узнали много нового о нашей системе. Мы увидели разные файлы и каталоги и их содержимое. Один из важнейших уроков, которые вы должны вынести, — это насколько открыта система. В Linux содержится много важных файлов, содержащих обычный легкочитаемый текст. В отличие от многих коммерческих систем, Linux полностью открыта для исследования и изучения.

4 Операции с файлами и каталогами

Теперь мы готовы приступить к настоящей работе! В этой главе будут представлены пять наиболее часто используемых команд Linux. Следующие команды используются для работы и с файлами, и с каталогами:

`cp` — копирует файлы и каталоги;

`mv` — перемещает/переименовывает файлы и каталоги;

`mkdir` — создает каталоги;

`rm` — удаляет файлы и каталоги;

`ln` — создает жесткие и символические ссылки.

Справедливости ради следует заметить, что некоторые задачи, выполняемые этими командами, гораздо проще решаются с помощью графического диспетчера файлов. В диспетчере файлов можно мышью перетаскивать файлы из одного каталога в другой, вырезать и вставлять файлы, удалять файлы и т. д. Но зачем тогда использовать эти старые программы командной строки?

Ответ прост: потому что они обладают мощностью и гибкостью. Несмотря на то что простые операции с файлами легко выполняются в диспетчере файлов с графическим интерфейсом, сложные задачи проще решать с помощью программ командной строки. Например, как скопировать файлы HTML из одного каталога в другой, причем только те, что отсутствуют в каталоге назначения или имеют более позднюю дату последнего изменения? Сделать это в диспетчере файлов очень сложно, но легко в командной строке:

```
cp -u *.html destination
```

Групповые символы

Прежде чем приступать к использованию обсуждаемых команд, необходимо сначала поговорить об одной особенности командной оболочки, которая делает эти команды такими мощными. Так как имена файлов используются в командной оболочке повсеместно, она поддерживает специальные символы, помогающие быстро определять группы имен файлов. Эти специальные символы называют *групповыми символами* (wildcards). Групповые символы (также известны как *символы подстановки* (globbing)) позволяют выбирать имена файлов по шаблону. В табл. 4.1 перечислены групповые символы и их соответствия.

Таблица 4.1. Групповые символы

Групповой символ	Соответствует
`*`	Любая последовательность любых символов
`?`	Любой один символ
`[символы]`	Любой один символ из указанного множества символов
`[!символы]`	Любой один символ, не принадлежащий указанному множеству символов
`[[:класс:]]`	Любой один символ, принадлежащий указанному классу

В табл. 4.2 перечислены наиболее часто используемые классы символов.

Таблица 4.2. Наиболее часто используемые классы символов

Класс символов	Соответствует
`[:alnum:]`	Любой алфавитно-цифровой символ
`[:alpha:]`	Любой алфавитный символ
`[:digit:]`	Любой цифровой символ
`[:lower:]`	Любая буква в нижнем регистре
`[:upper:]`	Любая буква в верхнем регистре

Групповые символы позволяют конструировать сложные критерии выбора имен файлов. В табл. 4.3 перечислены некоторые примеры шаблонов и их соответствия.

> **ГРУППОВЫЕ СИМВОЛЫ ТАКЖЕ ДЕЙСТВУЮТ В ГРАФИЧЕСКОМ ИНТЕРФЕЙСЕ**
>
> Групповые символы имеют особую ценность не только потому, что часто используются в командной строке, но и потому, что поддерживаются некоторыми диспетчерами с графическим интерфейсом.
>
> - В Nautilus (диспетчер файлов для GNOME) можно выбирать файлы с помощью диалога Edit (Правка) ► Select Pattern (Выделить по шаблону). Просто введите шаблон для выбора файлов с групповыми символами, и в текущем каталоге будут выделены файлы, соответствующие шаблону.
> - В некоторых версиях Dolphin и Konqueror (диспетчеры файлов для KDE) групповые символы можно вводить непосредственно в адресную строку. Например, если понадобится увидеть все файлы с именами, начинающимися с буквы «u» в нижнем регистре, в каталоге */usr/bin*, просто введите в адресной строке текст: `/usr/bin/u*`, и вы получите желаемый результат.
>
> Большинство идей, первоначально реализованных в интерфейсе командной строки, перекочевали и в графический интерфейс. Это одно из множества обстоятельств, которые делают настольный компьютер с Linux таким мощным инструментом.

Таблица 4.3. Примеры использования групповых символов

Шаблон	Соответствует
`*`	Все имена файлов
`g*`	Все имена файлов, начинающиеся с символа «g»
`b*.txt`	Все имена файлов, начинающиеся с символа «b», за которым следует любое число других символов, и заканчивающиеся на «.txt»
`Data???`	Все имена файлов, начинающиеся с символов «Data», за которыми следуют ровно три любых символа
`[abc]*`	Все имена файлов, начинающиеся с символа «a», «b» или «c»
`BACKUP.[0-9][0-9][0-9]`	Все имена файлов, начинающиеся с символов «BACKUP.», за которыми следуют ровно три цифровых символа
`[[:upper:]]*`	Все имена файлов, начинающиеся с буквы в верхнем регистре

Таблица 4.3 (*Окончание*)

Шаблон	Соответствует
`[![:digit:]]*`	Все имена файлов, не начинающиеся с цифры
`*[[:lower:]123]`	Все имена файлов, заканчивающиеся буквой в нижнем регистре или цифрой «1», «2» или «3»

Групповые символы можно использовать с любыми командами, принимающими имена файлов в виде аргументов, но подробнее об этом мы поговорим в главе 7.

ДИАПАЗОНЫ СИМВОЛОВ

Если у вас уже есть опыт работы с другим Unix-подобным окружением или вам приходилось читать другие книги по этой теме, вы встречали форму записи диапазонов символов `[A-Z]` или `[a-z]`. Это традиционные для Unix формы записи, и они прекрасно работают в старых версиях Linux. Более того, они все еще работают в новых версиях, но будьте очень осторожны при их использовании, потому что они не всегда дают ожидаемый результат без правильной настройки. А вообще, старайтесь избегать их и использовать классы символов.

mkdir — создание каталогов

Команда `mkdir` используется для создания каталогов. Вызывается она следующим образом:

```
mkdir каталог...
```

Примечание к форме записи: в этой книге три точки в описании команды, следующие за аргументом (как в примере выше), говорят о том, что аргументов может быть несколько; то есть в данном случае команда

```
mkdir dir1
```

создаст один каталог с именем *dir1*, а команда

```
mkdir dir1 dir2 dir3
```

создаст три каталога с именами *dir1*, *dir2* и *dir3*.

cp — копирование файлов и каталогов

Команда `cp` копирует файлы и каталоги. Ее можно использовать двумя разными способами:

```
cp item1 item2
```

чтобы скопировать один файл или каталог *item1* в файл или каталог *item2*, и

```
cp элемент... каталог
```

чтобы скопировать несколько элементов (файлов или каталогов) в указанный каталог.

Параметры команды cp и примеры ее использования

В табл. 4.4 перечислены некоторые параметры (короткие и эквивалентные им длинные), наиболее часто используемые с командой `cp`.

Таблица 4.4. Параметры команды cp

Параметр	Значение
`-a, --archive`	Скопировать файлы и каталоги со всеми атрибутами, включая идентификаторы владельцев и права доступа. Без этого параметра копии обычно получают значения атрибутов по умолчанию, определенных для пользователя, выполняющего копирование. Подробнее об атрибутах файлов рассказывается в главе 9
`-i, --interactive`	Запрашивать у пользователя подтверждение перед перезаписью существующего файла. **Если этот параметр отсутствует, команда** `cp` **просто перезапишет существующие файлы (без предупреждения)**
`-r, --recursive`	Рекурсивно копировать каталоги и их содержимое. Это обязательный параметр (или параметр `-a`) при копировании каталогов
`-u, --update`	При копировании файлов из одного каталога в другой копировать только файлы, отсутствующие в каталоге назначения или более новые. Этот параметр удобно использовать при копировании большого числа файлов, так как он позволяет пропустить файлы, которые не требуется копировать
`-v, --verbose`	Выводить информационные сообщения в процессе копирования

В табл. 4.5 приводится несколько примеров использования этих параметров.

Таблица 4.5. Примеры использования команды cp

Команда	Результат
`cp file1 file2`	Скопирует *file1* в *file2*. Если *file2* существует, он будет затерт новым файлом *file1*. Если *file2* отсутствует, он будет создан
`cp -i file1 file2`	То же, что и выше, но если файл *file2* существует, у пользователя будет запрошено подтверждение перед перезаписью файла
`cp file1 file2 dir1`	Скопирует *file1* и *file2* в каталог *dir1*. Каталог *dir1* должен существовать
`cp dir1/* dir2`	С использованием группового символа. Скопирует все файлы из каталога *dir1* в каталог *dir2*. Каталог *dir2* должен существовать
`cp -r dir1 dir2`	Скопирует содержимое каталога *dir1* в каталог *dir2*. Если каталог *dir2* не существует, он будет создан и заполнен содержимым каталога *dir1*. Если каталог *dir2* существует, тогда в него будет скопирован сам каталог *dir1* (вместе с его содержимым)

mv — перемещение и переименование файлов

Команда `mv` выполняет операции перемещения и переименования файлов в зависимости от особенностей использования. В любом случае исходный файл исчезает после операции. Команда `mv` используется почти так же, как команда `cp`:

```
mv item1 item2
```

перемещает или переименовывает файл или каталог *item1* в *item2*.

```
mv элемент... каталог
```

перемещает один или более элементов из одного каталога в другой.

Параметры команды mv и примеры ее использования

Команда `mv` поддерживает множество тех же параметров, что и команда `cp`, как показано в табл. 4.6.

Таблица 4.6. Параметры команды mv

Параметр	Значение
`-i, --interactive`	Запрашивать у пользователя подтверждение перед перезаписью существующего файла. **Если этот параметр отсутствует, команда `mv` просто перезапишет существующие файлы**
`-u, --update`	При перемещении файлов из одного каталога в другой перемещать только файлы, отсутствующие в каталоге назначения или более новые
`-v, --verbose`	Выводить информационные сообщения в процессе перемещения

В табл. 4.7 приводится несколько примеров использования команды `mv`.

Таблица 4.7. Примеры использования команды mv

Команда	Результат
`mv file1 file2`	Переместит *file1* в *file2*. **Если file2 существует, он будет заменен на новый файл file1.** Если *file2* отсутствует, он будет создан. **В любом случае файл file1 исчезнет**
`mv -i file1 file2`	То же, что и выше, но если файл *file2* существует, у пользователя будет запрошено подтверждение перед перезаписью файла
`mv file1 file2 dir1`	Переместит *file1* и *file2* в каталог *dir1*. Каталог *dir1* должен существовать
`mv dir1 dir2`	Если каталог *dir2* отсутствует, команда `mv` создаст его и переместит содержимое каталога *dir1* в каталог *dir2*, после чего удалит *dir1*. Если каталог *dir2* существует, команда `mv` переместит каталог *dir1* (вместе с его содержимым) в каталог *dir2*

rm — удаление файлов и каталогов

Команда `rm` используется для удаления (remove) файлов и каталогов, например:

```
rm элемент...
```

где `элемент` — это один или несколько файлов или каталогов.

Параметры команды rm и примеры ее использования

В табл. 4.8 перечислены некоторые параметры, наиболее часто используемые с командой `rm`.

Таблица 4.8. Параметры команды rm

Параметр	Значение
`-i, --interactive`	Запрашивать у пользователя подтверждение перед удалением существующего файла. **Если этот параметр отсутствует, команда `rm` просто удалит существующие файлы**
`-r, --recursive`	Рекурсивно удалить каталоги. То есть вместе с каталогом будут удалены все его подкаталоги. Это обязательный параметр при удалении каталогов
`-f, --force`	Игнорировать отсутствующие файлы и не запрашивать подтверждения. Этот параметр отменяет действие параметра `--interactive`
`-v, --verbose`	Выводить информационные сообщения в процессе удаления

БУДЬТЕ ОСТОРОЖНЫ С КОМАНДОЙ RM!

Unix-подобные операционные системы, такие как Linux, не имеют команды, отменяющей удаление. Если вы что-то удалили командой `rm`, это исчезнет навсегда. Linux считает вас достаточно ответственным человеком, отдающим себе отчет в своих действиях.

Будьте особенно осторожны с групповыми символами. Рассмотрим следующий классический пример. Допустим, вы захотели удалить все файлы HTML в каталоге. Для этого вы вводите команду:

```
rm *.html
```

которая сделает именно то, что вам нужно, но если вы случайно вставите пробел между `*` и `.html`, как в следующей команде:

```
rm * .html
```

`rm` удалит все файлы в каталоге и затем сообщит, что не нашла файла *.html*.

Полезный совет: всякий раз, используя групповые символы с командой `rm` (помимо внимательной проверки ввода!), проверьте сначала аргумент с групповым символом с командой `ls`. Это позволит увидеть, какие файлы будут удалены. Затем нажмите клавишу со стрелкой вверх, чтобы восстановить команду из истории, и замените `ls` на `rm`.

В табл. 4.9 приводится несколько примеров использования команды `rm`.

Таблица 4.9. Примеры использования команды rm

Команда	Результат
`rm file1`	Просто удалит файл *file1*
`rm -i file1`	Перед удалением *file1* запросит подтверждение у пользователя
`rm -r file1 dir1`	Удалит файл *file1* и каталог *dir1* со всем его содержимым
`rm -rf file1 dir1`	То же, что и выше, но в отсутствие *file1* и/или *dir1* просто продолжит работу, не выводя никаких сообщений

ln — создание ссылок

Команда `ln` применяется для создания жесткой или символической ссылки. Ее можно использовать одним из двух способов:

```
ln файл ссылка
```

создает жесткую ссылку.

```
ln -s элемент ссылка
```

создает символическую ссылку, где элементом может быть файл или каталог.

Жесткие ссылки

Жесткие ссылки — это первоначальный способ создания ссылок в Unix; символические ссылки — более позднее изобретение. По умолчанию каждый файл имеет одну жесткую ссылку, определяющую его имя. Создавая жесткую ссылку, мы создаем дополнительную запись в каталоге для файла. Жесткие ссылки имеют два важных ограничения.

- Жесткая ссылка не может указывать на файл за пределами собственной файловой системы. Это означает, что ссылка не может указывать на файл, находящийся в другом разделе диска.
- Жесткая ссылка не может указывать на каталог.

Жесткая ссылка неотличима от самого файла. В отличие от символических ссылок, при выводе списка с содержимым каталогов жесткие ссылки никак не выделяются. При удалении жесткой ссылки удаляется только сама ссылка, а файл остается на месте (то есть пространство, занимаемое файлом, не освобождается), пока не будут удалены все жесткие ссылки на файл.

Знать о существовании жестких ссылок важно, потому что они будут встречаться вам время от времени, но в современной практике предпочтение отдается символическим ссылкам, о которых рассказывается далее.

Символические ссылки

Символические ссылки были придуманы с целью преодолеть ограничения жестких ссылок. Когда создается символическая ссылка, в действительности создается файл особого типа, содержащий текстовый указатель на файл или каталог. В некотором отношении они действуют подобно ярлыкам в Windows, но, конечно же, появились задолго до ярлыков Windows.

Файл, на который указывает символическая ссылка, и сама символическая ссылка почти неотличимы друг от друга. Например, если попытаться что-то записать в символическую ссылку, запись будет выполнена в файл, на который она указывает. Однако при удалении символической ссылки удаляется *только символическая ссылка*, но *не файл*. Если удалить файл до того, как будет удалена символическая ссылка, ссылка останется на месте, но будет указывать в никуда. О таких ссылках говорят, что они *«битые»*. Во многих реализациях команда `ls` выделяет битые ссылки цветом, например красным, чтобы обратить на них внимание.

Идея ссылок может показаться странной и непонятной, но оставьте ее пока. Мы опробуем их на практике, и многое, возможно, для вас прояснится.

Постройка песочницы

Поскольку мы собираемся на практике опробовать некоторые операции с файлами, давайте выделим безопасный уголок для «игр» с командами управления файлами. Прежде всего нам понадобится каталог, в котором мы будем практиковаться. Создайте такой каталог в своем домашнем каталоге и назовите его `playground`.

Создание каталогов

Для создания каталогов используется команда `mkdir`. Чтобы создать каталог *playground*, проверьте сначала, находитесь ли вы в домашнем каталоге, и только потом создайте новый каталог:

```
[me@linuxbox ~]$ cd
[me@linuxbox ~]$ mkdir playground
```

Чтобы немножко украсить вашу песочницу, создайте внутри *playground* пару каталогов с именами *dir1* и *dir2*. Для этого смените текущий рабочий каталог на `playground` и выполните еще одну команду `mkdir`:

```
[me@linuxbox ~]$ cd playground
[me@linuxbox playground]$ mkdir dir1 dir2
```

Обратите внимание, что команда `mkdir` может принимать несколько аргументов, это позволяет создать два каталога одной командой.

Копирование файлов

Далее, добавим немного данных в нашу песочницу. Для этого скопируем какие-нибудь файлы. Командой `cp` скопируйте файл *passwd* из каталога */etc* в текущий рабочий каталог.

```
[me@linuxbox playground]$ cp /etc/passwd .
```

Обратите внимание на сокращение, обозначающее текущий рабочий каталог, — точку в конце команды. Если после этого выполнить команду `ls`, мы увидим наш файл:

```
[me@linuxbox playground]$ ls -l
итого 12
drwxrwxr-x 2 me me 4096 2018-01-10 16:40 dir1
drwxrwxr-x 2 me me 4096 2018-01-10 16:40 dir2
-rw-r--r-- 1 me me 1650 2018-01-10 16:07 passwd
```

Теперь ради развлечения повторите операцию копирования, но на этот раз с параметром `-v`, чтобы посмотреть, как она работает:

```
[me@linuxbox playground]$ cp -v /etc/passwd .
`/etc/passwd' -> `./passwd'
```

Команда `cp` вновь скопировала файл, но на этот раз вывела короткое сообщение, указывающее, что операция была выполнена. Обратите внимание, что `cp` переписала первую копию без каких-либо предупреждений. Это как раз тот случай, когда `cp` полагает, что вы знаете, что делаете. Чтобы вывести предупреждение, включите параметр `-i`:

```
[me@linuxbox playground]$ cp -i /etc/passwd .
cp: переписать `./passwd'?
```

Если в ответ на запрос ввести `y`, команда перезапишет существующий файл; если ввести любой другой символ (например, `n`), `cp` оставит прежнюю копию файла нетронутой.

Перемещение и переименование файлов

Имя *passwd* не выглядит органичным в нашей песочнице, поэтому дадим этому файлу какое-нибудь другое имя:

```
[me@linuxbox playground]$ mv passwd fun
```

Теперь немножко позабавимся и переместим переименованный файл в каждый из каталогов и обратно:

```
[me@linuxbox playground]$ mv fun dir1
```

переместит файл в каталог *dir1*. Следующая команда

```
[me@linuxbox playground]$ mv dir1/fun dir2
```

переместит файл из каталога *dir1* в каталог *dir2*. Следующая команда

```
[me@linuxbox playground]$ mv dir2/fun .
```

вернет его в текущий рабочий каталог. Теперь посмотрим, как `mv` влияет на каталоги. Сначала переместите файл в каталог *dir1*:

```
[me@linuxbox playground]$ mv fun dir1
```

Затем переместите *dir1* в *dir2* и проверьте их содержимое командой `ls`:

```
[me@linuxbox playground]$ mv dir1 dir2
[me@linuxbox playground]$ ls -l dir2
итого 4
drwxrwxr-x 2 me  me   4096 2018-01-11 06:06 dir1
[me@linuxbox playground]$ ls -l dir2/dir1
итого 4
-rw-r--r-- 1 me  me   1650 2018-01-10 16:33 fun
```

Обратите внимание: так как *dir2* уже существует, `mv` переместит *dir1* в *dir2*. Если бы каталога *dir2* не было, `mv` просто переименовала бы *dir1* в *dir2*. В заключение верните все на свои места:

```
[me@linuxbox playground]$ mv dir2/dir1 .
[me@linuxbox playground]$ mv dir1/fun .
```

Создание жестких ссылок

Теперь попробуем поиграть со ссылками. Сначала займемся жесткими ссылками: создайте несколько жестких ссылок для нашего файла:

```
[me@linuxbox playground]$ ln fun fun-hard
[me@linuxbox playground]$ ln fun dir1/fun-hard
[me@linuxbox playground]$ ln fun dir2/fun-hard
```

Теперь у нас есть четыре экземпляра файла *fun*. Посмотрим, что содержит наш каталог *playground*:

```
[me@linuxbox playground]$ ls -l
итого 16
drwxrwxr-x 2 me  me   4096 2018-01-14 16:17 dir1
drwxrwxr-x 2 me  me   4096 2018-01-14 16:17 dir2
-rw-r--r-- 4 me  me   1650 2018-01-10 16:33 fun
-rw-r--r-- 4 me  me   1650 2018-01-10 16:33 fun-hard
```

Прежде всего следует обратить внимание на второе поле в записях, соответствующих файлам *fun* и *fun-hard*. Оба содержат *4* — число жестких ссылок на файл, существующих в данный момент. Как вы помните, файл всегда имеет хотя бы одну жесткую ссылку, потому что имя файла определяется ссылкой. Но как убедиться, что *fun* и *fun-hard* — это один и тот же файл? В этом случае команда `ls` нам не помощник. Вы, конечно, скажете, что *fun* и *fun-hard* имеют одинаковые размеры (поле 5), но по списку файлов нельзя уверенно утверждать, что это один и тот же файл. Чтобы решить эту задачу, заглянем поглубже.

Рассуждая о жестких ссылках, полезно представлять файлы состоящими из двух частей:

- раздела с данными, где хранится содержимое файла;
- раздела с именем, где хранится имя файла.

Создавая жесткую ссылку, мы фактически создаем дополнительный раздел с именем, ссылающийся на тот же раздел с данными. Цепочку дисковых блоков система присваивает тому, что называется *индексным узлом* (inode), который затем присваивается разделу с именем. То есть каждая жесткая ссылка ссылается на определенный индексный узел с содержимым файла.

Команда `ls` может извлекать эту информацию. Для этого ее нужно вызвать с параметром `-i`:

```
[me@linuxbox playground]$ ls -li
итого 16
12353539 drwxrwxr-x 2 me  me   4096 2018-01-14 16:17 dir1
```

```
12353540 drwxrwxr-x 2 me  me   4096 2018-01-14 16:17 dir2
12353538 -rw-r--r-- 4 me  me   1650 2018-01-10 16:33 fun
12353538 -rw-r--r-- 4 me  me   1650 2018-01-10 16:33 fun-hard
```

В этой версии списка в первом поле отображается номер индексного узла, и, как можно видеть, оба имени, *fun* и *fun-hard*, ссылаются на индексные узлы с одним и тем же номером, а это подтверждает, что они соответствуют одному и тому же файлу.

Создание символических ссылок

Символические ссылки были придуманы с целью преодолеть ограничения жестких ссылок:

- жесткие ссылки не могут указывать на файлы, находящиеся на других физических устройствах;
- жесткие ссылки не могут указывать на каталоги — только на файлы.

Символическая ссылка — это файл особого типа, хранящий текстовый указатель на файл или каталог.

Создаются символические ссылки почти так же, как жесткие ссылки:

```
[me@linuxbox playground]$ ln -s fun fun-sym
[me@linuxbox playground]$ ln -s ../fun dir1/fun-sym
[me@linuxbox playground]$ ln -s ../fun dir2/fun-sym
```

Первый пример достаточно очевиден: мы просто добавили параметр `-s`, чтобы вместо жесткой ссылки создать символическую ссылку. Но два других выглядят несколько необычно. Не забывайте, что, создавая символическую ссылку, мы фактически определяем текст, описывающий местоположение целевого файла относительно символической ссылки. В этом легко убедиться, если взглянуть на вывод команды `ls`:

```
[me@linuxbox playground]$ ls -l dir1
итого 4
-rw-r--r-- 4 me  me   1650 2018-01-10 16:33 fun-hard
lrwxrwxrwx 1 me  me      6 2018-01-15 15:17 fun-sym -> ../fun
```

Запись с информацией о *fun-sym* в *dir1* сообщает, что это символическая ссылка (первый символ `l` в первом поле), указывающая на *../fun*, что правильно. Относительно символической ссылки *fun-sym* файл *fun* находится в каталоге уровнем выше. Обратите также внимание на размер файла символической ссылки,

равный 6, — это число символов в строке `../fun`, а не размер файла, на который она указывает.

При создании символических ссылок можно также указывать абсолютные пути, например:

```
[me@linuxbox playground]$ ln -s /home/me/playground/fun dir1/fun-sym
```

или относительные, как в более раннем примере. Но предпочтительнее использовать относительные пути, потому что это позволяет переименовывать и/или перемещать каталоги, содержащие символические ссылки, не разрушая их.

Помимо обычных файлов, символические ссылки могут указывать также на каталоги:

```
[me@linuxbox playground]$ ln -s dir1 dir1-sym
[me@linuxbox playground]$ ls -l
итого 16
drwxrwxr-x 2 me  me   4096 2018-01-15 15:17 dir1
lrwxrwxrwx 1 me  me      4 2018-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me  me   4096 2018-01-15 15:17 dir2
-rw-r--r-- 4 me  me   1650 2018-01-10 16:33 fun
-rw-r--r-- 4 me  me   1650 2018-01-10 16:33 fun-hard
lrwxrwxrwx 1 me  me      3 2018-01-15 15:15 fun-sym -> fun
```

Удаление файлов и каталогов

Как уже говорилось ранее, удаление файлов и каталогов выполняется при помощи команды `rm`. Далее мы немного почистим нашу песочницу. Сначала удалите одну из жестких ссылок:

```
[me@linuxbox playground]$ rm fun-hard
[me@linuxbox playground]$ ls -l
итого 12
drwxrwxr-x 2 me  me   4096 2018-01-15 15:17 dir1
lrwxrwxrwx 1 me  me      4 2018-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me  me   4096 2018-01-15 15:17 dir2
-rw-r--r-- 3 me  me   1650 2018-01-10 16:33 fun
lrwxrwxrwx 1 me  me      3 2018-01-15 15:15 fun-sym -> fun
```

Результат получился вполне ожидаемым. Файл *fun-hard* исчез, и счетчик ссылок во втором поле в записи для файла *fun* уменьшился с четырех до трех. Далее, удалите файл *fun* и ради развлечения добавьте в команду параметр `-i`, чтобы посмотреть, что происходит:

```
[me@linuxbox playground]$ rm -i fun
rm: удалить обычный файл `fun'?
```

Введите `y` в ответ на запрос, и файл будет удален. Но давайте посмотрим на вывод `ls`. Заметили, что произошло с *fun-sym*? Поскольку теперь символическая ссылка указывает на несуществующий файл, она стала битой:

```
[me@linuxbox playground]$ ls -l
итого 8
drwxrwxr-x 2 me  me   4096 2018-01-15 15:17 dir1
lrwxrwxrwx 1 me  me      4 2018-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me  me   4096 2018-01-15 15:17 dir2
lrwxrwxrwx 1 me  me      3 2018-01-15 15:15 fun-sym -> fun
```

В большинстве дистрибутивов Linux команда `ls` особым образом настраивается на отображение битых ссылок. Битые ссылки не представляют никакой опасности, но вносят определенную путаницу. При попытке использовать битую ссылку вы увидите:

```
[me@linuxbox playground]$ less fun-sym
fun-sym: Нет такого файла или каталога
```

Давайте немного приберем за собой. Удалите символическую ссылку:

```
[me@linuxbox playground]$ rm fun-sym dir1-sym
[me@linuxbox playground]$ ls -l
итого 8
drwxrwxr-x 2 me  me   4096 2018-01-15 15:17 dir1
drwxrwxr-x 2 me  me   4096 2018-01-15 15:17 dir2
```

Главное, что следует помнить о символических ссылках: большинство операций с файлами воздействуют на целевой элемент, а не на саму ссылку. Однако команда `rm` является исключением из этого правила. Когда вы удаляете ссылку, удаляется сама ссылка, а не элемент, на который она указывает.

В заключение удалим каталог *playground*. Для этого вернитесь в домашний каталог и вызовите команду `rm` с параметром рекурсивного удаления каталогов (`-r`), чтобы удалить каталог *playground* и все его содержимое, включая подкаталоги:

```
[me@linuxbox playground]$ cd
[me@linuxbox ~]$ rm -r playground
```

СОЗДАНИЕ СИМВОЛИЧЕСКИХ ССЫЛОК С ПОМОЩЬЮ ГРАФИЧЕСКОГО ИНТЕРФЕЙСА

Диспетчеры файлов в GNOME и KDE предоставляют простой автоматизированный способ создания символических ссылок. Если в GNOME во время перетаскивания файла мышью удерживать нажатыми клавиши CTRL и SHIFT, вместо копирования (или перемещения) файлов будет выполнена операция создания ссылки. В KDE, когда перетаскиваемый файл сбрасывается в целевой каталог, появляется небольшое меню, предлагающее выбор из трех операций: скопировать, переместить или создать ссылку.

Заключение

Мы узнали много нового, но чтобы информация усвоилась, требуется время. Выполняйте упражнения в песочнице раз за разом, пока не почувствуете, что понимаете их смысл. На данном этапе очень важно надежно усвоить, как работают основные команды управления файлами и групповые символы. Не бойтесь выйти за рамки предложенных упражнений — добавьте дополнительные файлы и каталоги, поэкспериментируйте с групповыми символами для определения групп файлов в разных операциях. Идея ссылок на первый взгляд может показаться малопонятной, поэтому уделите время их исследованию. Зачастую они оказываются настоящим спасательным кругом.

5 Работа с командами

До настоящего момента мы видели группы мистических команд, каждая из которых имеет свои таинственные параметры и аргументы. Теперь мы удалим часть этой таинственности и даже создадим несколько собственных команд. В этой главе будут представлены следующие команды:

`type` — сообщает, как интерпретируется имя указанной команды;

`which` — сообщает, какая программа будет выполнена;

`help` — выводит справку по встроенным командам оболочки;

`man` — выводит страницу справочного руководства с описанием команды;

`apropos` — выводит список подходящих команд;

`info` — выводит запись из справочного руководства *Info* с описанием команды;

`whatis` — выводит краткое описание команды;

`alias` — создает псевдоним для команды.

Что такое команды?

Команда может быть:

- **Выполняемой программой**, как те файлы, что мы видели в каталоге */usr/bin*. К этой категории относятся: *скомпилированные двоичные программы*, например, написанные на C и C++; программы, написанные на *языках сценариев*, таких как shell, Perl, Python, Ruby и др.
- **Встроенной командой, реализованной внутри самой командной оболочки.** Командная оболочка `bash` поддерживает множество внутренних команд, кото-

рые так и называют — *встроенными* (shell builtins). Команда `cd`, например, — это встроенная команда.

- **Функцией командной оболочки.** Функции командной оболочки (shell functions) — это миниатюрные сценарии на языке командной оболочки, встроенные в *окружение*. Мы еще вернемся к вопросам настройки окружения и создания функций командной оболочки в последующих главах, а пока просто помните об их существовании.
- **Псевдонимом.** Псевдоним (alias) — это команда, которую мы можем определить сами, сконструировав из других команд.

Идентификация команд

Часто бывает полезно точно знать, какому из четырех типов принадлежит команда, и Linux предлагает пару способов узнать это.

type — получение типа команды

Команда `type` — это встроенная команда, которая сообщает тип указанной ей команды. Вызывается она следующим образом:

```
type команда
```

где `команда` — это имя исследуемой команды. Например:

```
[me@linuxbox ~]$ type type
type встроена в оболочку
[me@linuxbox ~]$ type ls
ls является алиасом для `ls --color=tty'
[me@linuxbox ~]$ type cp
cp хэширована (/bin/cp)
```

Здесь мы видим результаты определения типов трех разных команд. Обратите внимание, что команда `ls` (в дистрибутиве Fedora) фактически является псевдонимом (alias) команды `ls` с параметром `--color=tty`. Теперь-то мы знаем, почему результаты команды `ls` отображаются в цвете!

which — определение местоположения выполняемого файла

Иногда в системе имеется более одной версии исполняемой программы. Это довольно редкое явление для настольных систем, но вполне обычное для больших

серверов. Точно определить местоположение данного исполняемого файла позволяет команда `which`:

```
[me@linuxbox ~]$ which ls
/bin/ls
```

`which` ищет только исполняемые программы, она не способна выявлять встроенные команды или псевдонимы, замещающие фактические исполняемые программы. Если попытаться с помощью `which` определить местоположение встроенной команды (например, `cd`), мы либо ничего не получим, либо получим сообщение об ошибке:

```
[me@linuxbox ~]$ which cd
/usr/bin/which: no cd in (/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games)
```

Это своеобразное сообщение «command not found» (команда не найдена).

Получение документации с описанием команд

Теперь, зная тип команды, можно поискать документацию с описанием, доступную для каждого вида команд[1].

help — получение справки для встроенных команд

`bash` имеет встроенную справку для каждой встроенной команды. Чтобы получить ее, введите `help` с именем встроенной команды. Например:

```
[me@linuxbox ~]$ help cd
cd: cd [-L|-P] [dir]
Change the current directory to DIR. The default DIR is the value of the
HOME shell variable.

The variable CDPATH defines the search path for the directory containing DIR.
Alternative directory names in CDPATH are separated by a colon (:). A null
directory name is the same as the current directory. If DIR begins with a slash
(/), then CDPATH is not used.

If the directory is not found, and the shell option `cdable_vars' is set, then try
the word as a variable name. If that variable has a value, its value is used for DIR.
```

[1] Некоторые разделы справки Linux переведены на русский язык, а некоторые — нет. Для переведенных разделов мы будем приводить русский текст, выводимый системой на консоль, для непереведенных — указывать перевод в сносках. — *Примеч. ред.*

```
Options:
-L force symbolic links to be followed: resolve symbolic links in DIR after
processing instances of `..'
-P use the physical directory structure without following symbolic links: resolve
symbolic links in DIR before processing instances of `..'
-e if the -P option is supplied, and the current working directory cannot be
determined successfully, exit with a non-zero status
-@ on systems that support it, present a file with extended attributes
   as a directory containing the file attributes

The default is to follow symbolic links, as if `-L' were specified. `..' is
processed by removing the immediately previous pathname component back to a slash
or the beginning of DIR.

Exit Status:
Returns 0 if the directory is changed, and if $PWD is set successfully when -P is
used; non-zero otherwise..[1]
```

[1] Перевод:

Делает указанный каталог `DIR` текущим.

Если каталог `DIR` не указан, по умолчанию используется значение переменной окружения `$HOME`.

Переменная `CDPATH` определяет пути поиска каталога, содержащего `DIR`. Альтернативные имена каталогов в `CDPATH` отделяются друг от друга двоеточием (`:`). Пустое имя каталога соответствует текущему каталогу. Если `DIR` начинается с символа слеш (`/`), переменная `CDPATH` не используется.

Если каталог не найден и установлен параметр `` `cdable_vars' `` командной оболочки, выполняется попытка интерпретировать слово как имя переменной. Если эта переменная имеет значение, тогда это значение используется как `DIR`.

Параметры:

`-L` требует следовать по символическим ссылкам: символические ссылки разрешаются в `DIR` после обработки элементов `` `..' ``

`–P` требует использовать физическую структуру каталогов без следования по символическим ссылкам: символические ссылки разрешаются в `DIR` до обработки элементов `` `..' ``

`-e` если указан параметр `-P` и текущий рабочий каталог невозможно определить, завершить команду с ненулевым кодом возврата

`-@` в системах, поддерживающих это, представляет файл с расширенными атрибутами как каталог, содержащий атрибуты файлов

По умолчанию следует по символическим ссылкам, как если бы был указан параметр `` `-L' ``. Встретив `` `..' ``, удаляет предыдущий компонент пути до ближайшего символа слеш или до начала `DIR`.

Возвращаемый код состояния:

Возвращает 0, если смена каталога произошла и переменной `$PWD` было благополучно присвоено значение (при использовании параметра `-P`); в остальных случаях возвращает ненулевое значение.

Примечание к форме записи: квадратные скобки в описании синтаксиса команды указывают на необязательность элемента. Вертикальная черта используется для перечисления взаимоисключающих вариантов. В примере с описанием команды `cd`, приведенном выше, ее синтаксис описывается как

```
cd [-L|[-P[-e]]] [dir]
```

Эта форма записи говорит, что команда `cd` может принимать необязательный параметр `-L` или `-P` и необязательный аргумент `dir`. Если указан параметр `-P`, вместе с ним можно также передать параметр `-e`.

Несмотря на то что `help` дает краткое и точное описание команды `cd`, это описание не может служить инструкцией по использованию, и, как вы можете видеть, в нем упоминается многое из того, чего мы еще не знаем! Но не волнуйтесь, со всем этим мы познакомимся в свое время.

--help — вывод инструкции по использованию

Многие выполняемые программы поддерживают параметр `--help` для вывода описания синтаксиса и параметров, поддерживаемых командой. Например:

```
[me@linuxbox ~]$ mkdir --help
Использование: mkdir [КЛЮЧ]... КАТАЛОГ...
Создает КАТАЛОГ(и), если он еще не существует.

  -Z, --context=CONTEXT (SELinux) установить контекст безопасности в CONTEXT
Аргументы, обязательные для длинных ключей, обязательны и для коротких.
  -m, --mode=РЕЖИМ установить режим доступа (как в chmod), не a=rwx – umask
  -p, --parents    не выдавать ошибок, если существует, создавать
                   родительские каталоги, если необходимо
  -v, --verbose    печатать сообщение о каждом созданном каталоге
      --help       показать эту справку и выйти
      --version    показать информацию о версии и выйти
Об ошибках в mkdir сообщайте по адресу <bug-coreutils@gnu.org>.
```

Некоторые программы не поддерживают параметр `--help`, но вы все равно пробуйте передать его. Часто в результате выводится сообщение об ошибке, содержащее ту же информацию о порядке использования.

man — вывод страниц справочного руководства

Большинство программ, предназначенных для использования в командной строке, предоставляют официальную документацию, которую называют *страницей*

справочного руководства (*man-страницу*). Для просмотра этих страниц используется специальная программа постраничного просмотра `man`, например:

```
man программа
```

где `программа` — имя команды.

Страницы справочного руководства могут несколько отличаться друг от друга оформлением, но в общем случае содержат:

- заголовок (название страницы);
- краткий обзор синтаксиса команды;
- описание назначения команды;
- список всех параметров с их описанием.

Однако страницы справочного руководства обычно не включают примеры использования, и их главная цель — служить справочником, а не инструкцией по использованию. Для примера попробуйте вывести страницу справочного руководства для команды `ls`:

```
[me@linuxbox ~]$ man ls
```

В большинстве систем Linux `man` использует `less` для вывода страницы, поэтому при просмотре страницы можно использовать все известные команды `less`.

«Руководство», которое отображает `man`, разбито на разделы и охватывает не только пользовательские команды, но и команды системного администрирования, программные интерфейсы, форматы файлов и многое другое. В табл. 5.1 перечислены разделы справочного руководства.

Таблица 5.1. Организация справочного руководства

Раздел	Содержит
1	Пользовательские команды
2	Программные интерфейсы системных вызовов в ядре
3	Программные интерфейсы в библиотеке C
4	Специальные файлы, такие как узлы устройств и драйверы
5	Форматы файлов
6	Игры и развлечения, такие как хранители экрана
7	Прочее
8	Команды системного администрирования

Иногда, чтобы найти искомое, нужно заглянуть в конкретный раздел. Это актуально для форматов файлов, названия которых часто совпадают с именами команд. Если номер раздела не указан, man всегда будет возвращать первую найденную страницу, обычно из раздела 1. Ниже приведен пример прямого указания номера раздела:

```
man раздел искомый_термин
```

Например:

```
[me@linuxbox ~]$ man 5 passwd
```

выведет страницу с описанием формата файла */etc/passwd*.

apropos — вывод списка подходящих команд

Кроме того, существует возможность найти страницы справочного руководства для близких совпадений с искомым термином. Несмотря на неточность, этот подход иногда оказывается полезным. Ниже приводится пример поиска страниц справочного руководства по слову *partition*:

```
[me@linuxbox ~]$ apropos partition
addpart (8)          - simple wrapper around the "add partition" ioctl
all-swaps (7)        - event signalling that all swap partitions have been ac...
cfdisk (8)           - display or manipulate disk partition table
cgdisk (8)           - Curses-based GUID partition table (GPT) manipulator
delpart (8)          - simple wrapper around the "del partition" ioctl
fdisk (8)            - manipulate disk partition table
fixparts (8)         - MBR partition table repair utility
gdisk (8)            - Interactive GUID partition table (GPT) manipulator
mpartition (1)       - partition an MSDOS hard disk
partprobe (8)        - inform the OS of partition table changes
partx (8)            - tell the Linux kernel about the presence and numbering...
resizepart (8)       - simple wrapper around the "resize partition" ioctl
sfdisk (8)           - partition table manipulator for Linux
sgdisk (8)           - Command-line GUID partition table (GPT) manipulator fo..[1]
```

[1] Перевод:

```
addpart (8)          - простая обертка для ioctl-команды "add partition"
all-swaps (7)        - событие, сигнализирующее, что все разделы подкачки были...
cfdisk (8)           - выводит или изменяет таблицу разделов диска
cgdisk (8)           - программа управления таблицами GUID разделов на основе curses
delpart (8)          - простая обертка для ioctl-команды "del partition"
fdisk (8)            - программа управления таблицей разделов диска
fixparts (8)         - утилита восстановления MBR таблицы разделов
gdisk (8)            - программа интерактивного управления таблицами GUID разделов (GPT)
```

Первое поле в каждой строке вывода — это имя страницы справочного руководства, а второе поле — номер раздела. Обратите внимание, что команда `man` с параметром `-k` действует как `apropos`.

> **САМАЯ БРУТАЛЬНАЯ СТРАНИЦА СПРАВОЧНОГО РУКОВОДСТВА**
>
> Как вы могли убедиться, страницы справочного руководства, входящие в состав Linux и других Unix-подобных систем, играют роль справочной документации, но не инструкций по использованию. Многие страницы очень сложно читать, но, как мне кажется, первый приз за сложность можно было бы присудить странице с описанием `bash`. Работая над книгой, я очень внимательно прочитал эту страницу, чтобы убедиться, что не упустил ни одной важной темы. Когда я ее распечатал, у меня получилось больше 80 страниц чрезвычайно плотного текста, структура которого не имеет никакого смысла для начинающих пользователей.
>
> С другой стороны, эта страница очень точная и краткая и содержит полную информацию. Поэтому почитайте ее, если у вас есть запас терпения, а затем ждите того момента, когда вы сможете читать ее и прочитанное будет наполнено для вас смыслом.

whatis — вывод очень краткого описания команды

Программа `whatis` выводит имя и однострочное описание из страницы справочного руководства, соответствующей искомому слову:

```
[me@linuxbox ~]$ whatis ls
ls                    (1) — выводит содержимое каталога
```

info — вывод записи из справочного руководства Info

В проекте GNU имеется альтернативное руководство Info, которое часто называют *info-страницами*. Info-страницы выводятся с помощью программы чтения

```
mpartition (1)       - создание разделов MSDOS на жестком диске
partprobe (8)        - информирует ОС об изменении таблицы разделов
partx (8)            - сообщает ядру Linux об имеющихся разделах и их нумерации...
resizepart (8)       - простая обертка для ioctl-команды "resize partition"
sfdisk (8)           - программа управления таблицами разделов для Linux
sgdisk (8)           - программа командной строки для управления таблицами GUID
                       разделов (GPT)...
```

с подходящим названием `info`. Info-страницы содержат гиперссылки, подобно веб-страницам. Например:

```
File: coreutils.info, Node: ls invocation, Next: dir invocation, Up:
Directory listing
10.1 `ls': List directory contents
==================================
The `ls' program lists information about files (of any type, including
directories). Options and file arguments can be intermixed arbitrarily, as
usual.
For non-option command-line arguments that are directories, by default `ls'
lists the contents of directories, not recursively, and omitting files with
names beginning with `.'. For other non-option arguments, by default `ls'
lists just the filename. If no non-option argument is specified, `ls' operates
on the current directory, acting as if it had been invoked with a single
argument of `.'.
By default, the output is sorted alphabetically, according to the
--zz-Info: (coreutils.info.gz)ls invocation, 63 lines --Top----------[1]
```

Программа `info` читает info-файлы, организованные в древовидную структуру отдельных *узлов*, каждый из которых содержит отдельную тему. Info-файлы включают гиперссылки, с помощью которых можно перемещаться от узла к узлу. Гиперссылку можно узнать по начальному символу звездочки. Гиперссылки активируются при установке текстового курсора на них и осуществляют переход при нажатии клавиши ENTER.

Чтобы вывести info-страницу, введите команду `info` и добавьте после нее необязательное имя интересующей программы. В табл. 5.2 перечислены команды, которые можно использовать для управления программой во время чтения info-страницы.

[1] Перевод:

```
Файл: coreutils.info, Узел: команда ls, Следующий: команда dir, Up:
Содержимое каталога
10.1 `ls': выводит содержимое каталога
==================================
Программа `ls' выводит информацию о файлах (любого типа, включая
каталоги). Параметры и аргументы могут смешиваться произвольно, как
обычно.
Для аргументов без параметров, представляющих каталоги, по умолчанию `ls'
выводит содержимое каталогов нерекурсивно и пропускает файлы с
именами, начинающимися с `.'. Для других аргументов без параметров по умолчанию `ls'
выводит только указанный файл. В отсутствие аргументов без параметров `ls'
выводит содержимое текущего каталога, действуя, как если бы была вызвана с
единственным аргументом `.'.
По умолчанию вывод сортируется в алфавитном порядке, согласно
--zz-Info: (coreutils.info.gz)команда ls, 63 строки --Top----------
```

Таблица 5.2. Команды программы info

Команда	Действие
?	Вывести справку
Page Up или Backspace	Вывести предыдущую страницу
Page Down или ПРОБЕЛ	Вывести следующую страницу
n	Вперед (next) — вывести следующий узел
p	Назад (previous) — вывести предыдущий узел
u	Вверх (up) — вывести узел, родительский по отношению к текущему, обычно меню
ENTER	Перейти по гиперссылке, находящейся на позиции курсора
q	Завершить (quit)

Большинство программ из числа рассматривавшихся до сих пор, является частью пакета coreutils проекта GNU, поэтому о них можно получить дополнительную информацию командой

```
[me@linuxbox ~]$ info coreutils
```

Она выведет страницу с меню, состоящим из гиперссылок на документацию для каждой программы, входящей в состав пакета coreutils.

README и другие файлы с описанием программ

Многие программные пакеты, установленные в вашей системе, включают файлы с документацией, размещаемые в каталоге */usr/share/doc*. Большинство из них имеют простой текстовый формат и могут просматриваться с помощью `less`. Некоторые файлы имеют формат HTML и могут просматриваться с помощью веб-браузера. Можно также встретить файлы с расширением *.gz*. Это сжатые файлы, обработанные программой-архиватором `gzip`. Пакет `gzip` включает специальную версию `less` с именем `zless`, которая выводит содержимое текстовых файлов, сжатых архиватором `gzip`.

Создание собственных команд с помощью alias

А теперь проведем первый опыт по программированию! У нас есть возможность создавать собственные команды с помощью команды `alias`. Но прежде чем начать,

познакомимся с одной маленькой хитростью командной строки. Она позволяет уместить в одной строке несколько команд, для чего нужно просто отделить их друг от друга точкой с запятой:

```
команда1; команда2; команда3...
```

Следующий пример демонстрирует этот прием:

```
[me@linuxbox ~]$ cd /usr; ls; cd -
bin games include lib local sbin share src
/home/me
[me@linuxbox ~]$
```

Как видите, мы поместили три команды в одну строку. Первая выполняет переход в каталог */usr*, вторая выводит его содержимое, и третья осуществляет возврат в предыдущий каталог (команда `cd -`), поэтому по завершении мы оказываемся там же, где и были. Давайте теперь с помощью `alias` превратим эту последовательность в новую команду. Первое, что мы должны сделать, — придумать имя для новой команды. Пусть это будет `test`. Но прежде чем продолжить, хорошо бы проверить, не занято ли уже имя `test`. Для этого воспользуемся командой `type`:

```
[me@linuxbox ~]$ type test
test встроена в оболочку
```

Ой! Имя `test` уже занято. Попробуем `foo`:

```
[me@linuxbox ~]$ type foo
bash: type: foo: не найден
```

Отлично! Имя `foo` свободно. Теперь создадим наш псевдоним:

```
[me@linuxbox ~]$ alias foo='cd /usr; ls; cd -'
```

Обратите внимание на структуру этой команды:

```
alias имя='строка'
```

За командой `alias` следует имя, сразу за которым (то есть без пробелов) следует знак «равно» и строка в кавычках, описывающая действие, присваиваемое имени. После определения псевдонима его можно использовать везде, где ожидается команда. Давайте попробуем:

```
[me@linuxbox ~]$ foo
bin games inclide lib local sbin share src
/home/me
[me@linuxbox ~]$
```

Команда `type` правильно определяет псевдонимы:

```
[me@linuxbox ~]$ type foo
foo является алиасом для `cd /usr; ls; cd -'
```

Удалить псевдоним можно с помощью команды `unalias`:

```
[me@linuxbox ~]$ unalias foo
[me@linuxbox ~]$ type foo
bash: type: foo: не найден
```

Несмотря на то что в этом примере мы постарались не использовать имя существующей команды, иногда это бывает полезно. Часто это делается, чтобы применить наиболее желательные параметры к каждому вызову команды. Например, выше мы видели, что команда `ls` часто определяется как псевдоним, — это позволяет реализовать вывод информации в цвете:

```
[me@linuxbox ~]$ type ls
ls является алиасом для `ls --color=tty'
```

Если вызвать команду `alias` без аргументов, она выведет список всех псевдонимов в окружении. Ниже приводятся несколько псевдонимов, объявляемых в дистрибутиве Fedora по умолчанию. Попробуйте понять, что они делают:

```
[me@linuxbox ~]$ alias
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
```

Существует одна маленькая проблема, связанная с определением псевдонимов в командной строке. Они исчезают по завершении сеанса работы с командной оболочкой. В главе 11 будет показано, как добавить определения псевдонимов в файлы, чтобы они восстанавливались при каждом запуске командной оболочки, а пока насладимся нашим первым, пусть и крошечным, шагом в мир программирования на языке командной оболочки!

Заключение

Теперь, когда мы узнали, как найти документацию с описанием команд, поупражняйтесь самостоятельно и найдите описание всех команд, встретившихся вам в этой книге. Познакомьтесь с их дополнительными параметрами и опробуйте их!

6 Перенаправление

В этом уроке мы познакомимся с самой крутой возможностью командной строки: *перенаправлением ввода/вывода*. Благодаря этой возможности мы сможем перенаправлять ввод и вывод команд из файлов и в файлы, а также составлять из команд целые *конвейеры*. Для демонстрации этой возможности введем в обиход следующие команды:

`cat` — объединяет файлы;

`sort` — сортирует строки текста;

`uniq` — сообщает о повторяющихся строках или удаляет их;

`grep` — находит и выводит строки, соответствующие шаблону;

`wc` — выводит число символов перевода строки, слов и байтов в каждом указанном файле;

`head` — выводит первые строки из файла;

`tail` — выводит последние строки из файла;

`tee` — читает данные со стандартного ввода и записывает в стандартный вывод и в файлы.

Стандартный ввод, вывод и вывод ошибок

Многие программы, которыми мы уже пользовались, что-нибудь выводят на консоль. Этот вывод часто делится на два типа:

- Результаты работы программы, то есть данные, для получения которых создавалась программа.

- Сообщения о состоянии или об ошибках, извещающие нас о самочувствии программы.

Например, если взглянуть на вывод программы `ls`, можно увидеть, что она выводит на экран результаты своей работы и иногда сообщения об ошибках.

Согласно центральной идее Unix, что «все сущее есть файл», такие программы, как `ls`, в действительности выводят свои результаты в специальный файл, который называется *стандартным выводом* (standard output, часто обозначается как *stdout*), а сообщения о состоянии — в специальный файл *стандартный вывод ошибок* (standard error, *stderr*). По умолчанию оба файла, стандартный вывод и стандартный вывод ошибок, связаны с экраном и не сохраняются на диске.

Кроме того, многие программы принимают ввод из специального файла с названием *стандартный ввод* (standard input, *stdin*), который по умолчанию связан с клавиатурой.

Механизм перенаправления ввода/вывода позволяет изменять направление вывода и ввода. Обычно вывод осуществляется на экран, а ввод — с клавиатуры, но механизм перенаправления ввода/вывода позволяет изменить этот порядок вещей.

Перенаправление стандартного вывода

Механизм перенаправления ввода/вывода позволяет явно указать, куда должен осуществляться стандартный вывод. Чтобы перенаправить стандартный вывод в другой файл вместо экрана, нужно добавить в команду оператор перенаправления `>` и имя файла. Где это может пригодиться? Иногда полезно сохранить вывод команды в файл. Например, можно сообщить командной оболочке, что она должна направить вывод команды `ls` в файл *ls-output.txt* вместо экрана:

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Здесь мы создали длинный список содержимого каталога */usr/bin* и отправили результаты в файл *ls-output.txt*. Давайте исследуем перенаправленный вывод команды:

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me   me   167878 2018-02-01 15:07 ls-output.txt
```

Неплохой файл получился. Если вывести содержимое *ls-output.txt* с помощью команды `less`, можно увидеть, что он действительно содержит результаты работы команды `ls`:

```
[me@linuxbox ~]$ less ls-output.txt
```

Давайте теперь повторим эксперимент с перенаправлением, но с небольшим усложнением: укажем имя несуществующего каталога:

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt
ls: невозможно получить доступ к '/bin/usr': Нет такого файла или каталога
```

Мы получили сообщение об ошибке. Все логично — мы указали несуществующий каталог */bin/usr*, но почему же сообщение появилось на экране, а не было перенаправлено в файл *ls-output.txt*? Дело в том, что программа `ls` не выводит сообщения об ошибках в стандартный вывод. Как и многие добропорядочные программы для Unix, она выводит сообщения об ошибках в стандартный поток вывода ошибок. Поскольку мы перенаправили только стандартный вывод, а стандартный вывод ошибок — нет, сообщение об ошибке появилось на экране. Как перенаправить стандартный вывод ошибок, будет показано чуть ниже, но перед этим посмотрим, что произошло с нашим файлом:

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me   me   0 2018-02-01 15:08 ls-output.txt
```

Файл очистился! Это объясняется тем, что при перенаправлении вывода с помощью оператора `>` файл назначения всегда перезаписывается с самого начала. Поскольку команда `ls` не вывела никаких результатов, а только сообщение об ошибке, оператор перенаправления перезаписал файл, а затем остановился из-за ошибки, что привело к его очистке. Получается, что если вам понадобится очистить какой-нибудь файл (или создать новый, пустой файл), это можно сделать с помощью следующего трюка:

```
[me@linuxbox ~]$ > ls-output.txt
```

Простой оператор перенаправления, без предшествующей ему команды, очистит существующий файл или создаст новый, пустой файл.

Так как же добавить вывод в конец существующего файла, не затерев его? Для этого используем оператор перенаправления `>>`:

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
```

Оператор `>>` просто добавит результаты в конец файла. Если файл не существует, он будет создан, как при использовании оператора `>`. Давайте протестируем его:

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
```

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me   me   503634 2018-02-01 15:45 ls-output.txt
```

Мы повторили команду трижды и получили файл втрое большего размера.

Перенаправление стандартного вывода ошибок

Перенаправление стандартного вывода ошибок осуществляется не так просто, как стандартного вывода. Чтобы перенаправить стандартный вывод ошибок, нужно указать его *дескриптор файла*. Программа может производить вывод в любой из нескольких нумерованных файловых потоков. Первые три из них мы упомянули как стандартный ввод, вывод и вывод ошибок. Командная оболочка ссылается на них как на файловые дескрипторы 0, 1 и 2 соответственно. Командная оболочка поддерживает синтаксис перенаправления файлов с использованием номеров файловых дескрипторов. Так как стандартному выводу ошибок соответствует файловый дескриптор 2, мы можем перенаправить его, как показано ниже:

```
[me@linuxbox ~]$ ls -l /bin/usr 2> ls-error.txt
```

Номер файлового дескриптора 2 помещается непосредственно перед оператором перенаправления, чтобы перенаправить стандартный вывод ошибок в файл *ls-error.txt*.

Перенаправление стандартного вывода и стандартного вывода ошибок в один файл

Иногда необходимо сохранить весь вывод команды в один файл. Для этого перенаправьте сразу два потока, стандартный вывод и стандартный вывод ошибок. Сделать это можно двумя способами. Первый — традиционный — работает в старых версиях командной оболочки:

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt 2>&1
```

Здесь выполняются два перенаправления. Сначала — перенаправление стандартного вывода в файл *ls-output.txt*, а затем, с использованием нотации `2>&1`, — перенаправление файлового дескриптора 2 (стандартный вывод ошибок) в файловый дескриптор 1 (стандартный вывод).

> **ИМЕЙТЕ В ВИДУ, ЧТО ПОРЯДОК ПЕРЕНАПРАВЛЕНИЯ ИГРАЕТ ВАЖНУЮ РОЛЬ**
>
> Перенаправление стандартного вывода ошибок всегда должно производиться *после* перенаправления стандартного вывода, иначе этот трюк не сработает. Следующий пример перенаправит стандартный вывод ошибок в файл *ls-output.txt*:
>
> ```
> > ls-output.txt 2>&1
> ```
>
> Если порядок перенаправления изменить, как показано ниже, стандартный вывод ошибок будет перенаправлен на экран:
>
> ```
> 2>&1 >ls-output.txt
> ```

Современные версии bash поддерживают второй, более простой метод выполнения перенаправления этого вида:

```
[me@linuxbox ~]$ ls -l /bin/usr &> ls-output.txt
```

В данном примере используется единственный оператор `&>`, перенаправляющий стандартный вывод и стандартный вывод ошибок в файл *ls-output.txt*. Аналогично можно перенаправить стандартный вывод и стандартный вывод ошибок с добавлением в конец одного и того же файла:

```
[me@linuxbox ~]$ ls -l /bin/usr &>> ls-output.txt
```

Удаление нежелательного вывода

Иногда молчание действительно золото, и вывод команды нужно отбросить. В особенности это касается служебных сообщений и сообщений об ошибках. Система дает такую возможность, предоставляя специальный файл */dev/null*, куда можно перенаправить вывод. Этот файл представляет системное устройство, называемое *битоприемником* (bit bucket) или мусорной корзиной, которое принимает любой ввод и ничего с ним не делает. Чтобы подавить вывод сообщений об ошибках, достаточно проделать следующее:

```
[me@linuxbox ~]$ ls -l /bin/usr 2> /dev/null
```

> **/DEV/NULL В КУЛЬТУРЕ UNIX**
>
> «Битоприемник» — старое понятие в Unix, благодаря своему универсализму широко используется в культуре Unix. Так, когда кто-то скажет, что посылает ваши комментарии в */dev/null*, вы теперь будете знать, что это означает. Еще больше примеров вы найдете в статье Википедии *https://ru.wikipedia.org/wiki//dev/null*.

Перенаправление стандартного ввода

До сих пор нам не встречались команды, использующие стандартный ввод (на самом деле они встречались, но мы подробнее обсудим их чуть ниже), поэтому нам нужно познакомиться с ними.

cat — объединение файлов

Команда `cat` читает содержимое одного или нескольких файлов и копирует его в стандартный вывод:

```
cat [файл...]
```

Часто команду `cat` можно считать аналогом команды `TYPE` в DOS. Она используется для вывода содержимого файлов без возможности постраничного просмотра. Например,

```
[me@linuxbox ~]$ cat ls-output.txt
```

выведет содержимое файла *ls-output.txt*.

Команда `cat` часто используется для вывода коротких текстовых файлов. Поскольку `cat` способна принимать сразу несколько файлов, она используется для их объединения. Представьте, что вы загрузили большой файл, разбитый на множество частей (в Usenet мультимедийные файлы часто разбиваются таким способом), и требуется объединить их в один файл. Если файлы имеют имена, такие как

```
movie.mpeg.001 movie.mpeg.002 ... movie.mpeg.099
```

их можно объединить следующей командой:

```
[me@linuxbox ~]$ cat movie.mpeg.0* > movie.mpeg
```

Поскольку подстановка фактических имен взамен *групповых символов* всегда выполняется в порядке сортировки, аргументы окажутся расположенными в правильном порядке.

Все это прекрасно, но при чем здесь стандартный ввод? Пока ни при чем, но давайте попробуем кое-что еще. Что получится, если вызвать `cat` без аргументов?

```
[me@linuxbox ~]$ cat
```

Ничего не произошло — такое ощущение, что команда зависла. Однако в действительности команда делает именно то, что и предполагалось.

Если вызвать cat без аргументов, она начнет читать данные со стандартного ввода, а поскольку стандартный ввод по умолчанию подключен к клавиатуре, получается, что команда ждет, пока вы что-нибудь напечатаете! Попробуйте так:

```
[me@linuxbox ~]$ cat
Съешь ещё этих мягких французских булок, да выпей чаю.
```

Затем нажмите комбинацию CTRL+D (то есть, удерживая нажатой клавишу CTRL, нажмите клавишу D), чтобы сообщить команде cat, что достигнут конец файла (end-of-file, EOF) на стандартном вводе:

```
[me@linuxbox ~]$ cat
Съешь ещё этих мягких французских булок, да выпей чаю.
Съешь ещё этих мягких французских булок, да выпей чаю.
```

В отсутствие аргументов с именами файлов cat копирует содержимое стандартного ввода в стандартный вывод, поэтому-то мы и увидели, как она повторила введенную нами строку. Эту ее особенность можно использовать для создания коротких текстовых файлов. Представьте, что вам потребовалось создать файл с именем *eat_more.txt*, содержащий текст из примера, приведенного выше. Сделать это можно было бы так:

```
[me@linuxbox ~]$ cat > eat_more.txt
Съешь ещё этих мягких французских булок, да выпей чаю.
```

Введите команду, затем текст, который нужно поместить в файл, и не забудьте нажать комбинацию CTRL+D в конце. Используя командную строку, мы реализовали самый простой в мире текстовый процессор! Чтобы увидеть результат, воспользуемся командой cat и скопируем файл в стандартный вывод:

```
[me@linuxbox ~]$ cat eat_more.txt
Съешь ещё этих мягких французских булок, да выпей чаю.
```

Теперь, когда мы знаем, что команда cat может принимать данные не только из файлов, указанных в аргументах, но и со стандартного ввода, попробуем выполнить перенаправление стандартного ввода:

```
[me@linuxbox ~]$ cat < eat_more.txt
Съешь ещё этих мягких французских булок, да выпей чаю.
```

Используя оператор перенаправления <, мы изменили источник данных для стандартного ввода с клавиатуры на файл *eat_more.txt*. Как видите, результат получился тот же, как если бы мы просто передали единственный аргумент с именем файла. Этот способ не имеет никаких преимуществ в сравнении с передачей простого

аргумента, но он демонстрирует, как можно использовать файлы в роли источника данных для стандартного ввода. Другие команды находят лучшее применение стандартному вводу, в чем мы вскоре убедимся.

Прежде чем двинуться дальше, прочитайте страницу справочного руководства (man) для команды `cat`, так как она имеет несколько очень интересных параметров.

Конвейеры

«Умение» команд читать данные со стандартного ввода и выводить результаты в стандартный вывод используется механизмом командной оболочки, который называется *конвейером*. С помощью оператора конвейера[1] | (вертикальная черта) стандартный вывод одной команды можно связать со стандартным вводом другой.

```
команда1 | команда2
```

Для демонстрации этого механизма нам понадобится несколько команд. Мы уже упоминали команду, которая может получать данные со стандартного ввода. Это команда `less`. Теперь используем `less` для постраничного отображения вывода любой команды, которая посылает свои результаты в стандартный вывод:

```
[me@linuxbox ~]$ ls -l /usr/bin | less
```

Это очень удобно! С помощью этого приема можно со всем комфортом исследовать вывод любой команды, посылающей результаты на стандартный вывод.

Фильтры

Конвейеры часто используются для выполнения сложных операций с данными. Они позволяют объединить вместе несколько команд. Часто команды, используемые таким способом, называют *фильтрами*. Фильтры принимают ввод, изменяют его определенным образом и выводят результат. Первый из таких фильтров, который мы опробуем, — команда `sort`. Представьте, что нам необходимо составить список всех выполняемых программ в каталогах */bin* и */usr/bin*, расположив их по алфавиту, и затем вывести его:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | less
```

[1] Часто этот оператор называют также оператором канала. — *Примеч. пер.*

Поскольку в команде указаны два каталога (*/bin* и */usr/bin*), вывод команды `ls` будет состоять из двух сортированных списков, по одному для каждого каталога. Добавив команду `sort` в конвейер, мы изменили данные, чтобы получить единый сортированный список.

> **РАЗЛИЧИЯ МЕЖДУ ≥ И |**
>
> С первого взгляда трудно понять разницу между оператором конвейера | и оператором перенаправления >. Выражаясь простым языком, оператор перенаправления связывает команду с файлом, а оператор конвейера связывает вывод одной команды с вводом другой.
>
> ```
> команда1 > файл1
> команда1 | команда2
> ```
>
> Многие, впервые познакомившись с идеей конвейера, пробуют проделать следующий трюк, «только чтобы посмотреть, что из этого получится»:
>
> ```
> команда1 > команда2
> ```
>
> Не повторяйте этот эксперимент: иногда он может стать причиной больших проблем.
>
> Вот конкретный пример, представленный читателем, который по роду своей деятельности занимается администрированием сервера с ОС Linux. Зарегистрировавшись как суперпользователь, он выполнил следующую пару команд:
>
> ```
> # cd /usr/bin
> # ls > less
> ```
>
> Первая команда выполнила переход в каталог, где хранится большинство программ, а вторая потребовала от командной оболочки очистить файл *less* и записать в него вывод команды `ls`. Так как в каталоге */usr/bin* уже имеется файл с именем *less* (программа `less`), вторая команда затерла программу в файле *less* текстом, который вернула команда `ls`, и тем самым уничтожила программу `less`.
>
> Помните, что оператор перенаправления без лишних предупреждений создает или затирает файлы, поэтому относитесь к нему с особым вниманием.

uniq — поиск или удаление повторяющихся строк

Команда `uniq` часто используется в комбинации с командой `sort`. `uniq` принимает сортированный список данных либо со стандартного ввода, либо из файла, имя которого можно передать в единственном аргументе (за подробностями

обращайтесь к странице справочного руководства (man) для команды `uniq`), и по умолчанию удаляет повторяющиеся строки из списка. Поэтому, чтобы гарантировать отсутствие дубликатов в нашем списке (то есть любых программ с одинаковыми именами в каталогах */bin* и */usr/bin*), добавим `uniq` в конвейер:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | less
```

В этом примере мы использовали `uniq` для удаления любых повторяющихся строк в выводе команды `sort`. Если бы нам потребовалось, наоборот, получить список дубликатов, мы добавили бы в команду `uniq` параметр `-d`:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq -d | less
```

wc — вывод числа строк, слов и байтов

Команда `wc` (word count — счетчик слов) используется для подсчета числа строк, слов и байтов в файлах. Например:

```
[me@linuxbox ~]$ wc ls-output.txt
7902   64566 503634 ls-output.txt
```

В данном случае команда вывела три числа: число строк, число слов и число байтов в файле *ls-output.txt*. Подобно предыдущим командам, она может вызываться без аргументов, и в этом случае `wc` будет принимать данные со стандартного ввода. Параметр `-l` ограничивает вывод результатов только числом строк. Команду удобно использовать в конвейерах для подсчета: например, подсчитать число элементов в нашем сортированном списке можно так:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | wc -l
2728
```

grep — поиск строк, соответствующих шаблону

`grep` — очень мощная программа, она часто используется для поиска в файлах текста по шаблону:

```
grep шаблон [файл...]
```

Когда `grep` находит в файле совпадение с «шаблоном», она выводит строку с найденным совпадением. Шаблоны, используемые командой `grep` для поиска, могут быть очень сложными, но сейчас мы рассмотрим только поиск прямого совпадения с текстом. Более сложные шаблоны, которые называют *регулярными выражениями*, мы рассмотрим в главе 19.

Допустим, что нам нужно найти все файлы в списке программ, которые имеют в своем имени последовательность символов *zip*. Результаты такого поиска могут подсказать нам, какие программы в системе имеют отношение к сжатию файлов. Сделать это можно так:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | grep zip
bunzip2
bzip2
gunzip
gzip
unzip
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
```

Команда `grep` имеет пару удобных параметров:

- `-i` требует от `grep` игнорировать регистр символов в процессе поиска (обычно поиск выполняется с учетом регистра символов);
- `-v` требует от `grep` выводить только строки, где совпадение с шаблоном не найдено.

head/tail — вывод первых/последних строк из файлов

Иногда требуется выводить не все результаты работы команды, а только несколько первых или несколько последних строк. Команда `head` выводит первые 10 строк из файла, а `tail` — последние 10 строк. По умолчанию обе команды выводят 10 строк текста, но это число можно изменить с помощью параметра `-n`:

```
[me@linuxbox ~]$ head -n 5 ls-output.txt
total 343496
-rwxr-xr-x 1 root root      31316 2017-12-05 08:58 [
-rwxr-xr-x 1 root root       8240 2017-12-09 13:39 411toppm
-rwxr-xr-x 1 root root     111276 2017-11-26 14:27 a2p
-rwxr-xr-x 1 root root      25368 2016-10-06 20:16 a52dec
[me@linuxbox ~]$ tail -n 5 ls-output.txt
-rwxr-xr-x 1 root root       5234 2017-06-27 10:56 znew
-rwxr-xr-x 1 root root        691 2005-09-10 04:21 zonetab2pot.py
-rw-r--r-- 1 root root        930 2017-11-01 12:23 zonetab2pot.pyc
-rw-r--r-- 1 root root        930 2017-11-01 12:23 zonetab2pot.pyo
lrwxrwxrwx 1 root root          6 2016-01-31 05:22 zsoelim -> soelim
```

Их также можно использовать в конвейерах:

```
[me@linuxbox ~]$ ls /usr/bin | tail -n 5
znew
zonetab2pot.py
zonetab2pot.pyc
zonetab2pot.pyo
zsoelim
```

Команда `tail` позволяет наблюдать, как изменяется содержимое файла в режиме реального времени. Эту ее особенность удобно использовать для наблюдения за появлением новых записей в файлах журналов. В следующем примере демонстрируется наблюдение за файлом *messages* в каталоге */var/log*. В некоторых дистрибутивах Linux для этого требуется обладать привилегиями суперпользователя, поскольку файл */var/log/messages* может содержать секретную информацию.

```
[me@linuxbox ~]$ tail -f /var/log/messages
Feb  8 13:40:05 twin4 dhclient: DHCPACK from 192.168.1.1
Feb  8 13:40:05 twin4 dhclient: bound to 192.168.1.4 -- renewal in 1652 seconds.
Feb  8 13:55:32 twin4 mountd[3953]: /var/NFSv4/musicbox exported to both
192.168.1.0/24 and twin7.localdomain in 192.168.1.0/24,twin7.localdomain
Feb  8 14:07:37 twin4 dhclient: DHCPREQUEST on eth0 to 192.168.1.1 port 67
Feb  8 14:07:37 twin4 dhclient: DHCPACK from 192.168.1.1
Feb  8 14:07:37 twin4 dhclient: bound to 192.168.1.4 -- renewal in 1771 seconds.
Feb  8 14:09:56 twin4 smartd[3468]: Device: /dev/hda, SMART Prefailure Attribute:
8 Seek_Time_Performance changed from 237 to 236
Feb  8 14:10:37 twin4 mountd[3953]: /var/NFSv4/musicbox exported to both
192.168.1.0/24 and twin7.localdomain in 192.168.1.0/24,twin7.localdomain
Feb  8 14:25:07 twin4 sshd(pam_unix)[29234]: session opened for user me by (uid=0)
Feb  8 14:25:36 twin4 su(pam_unix)[29279]: session opened for user root by
me(uid=500)
```

При вызове с параметром `-f` команда `tail` продолжает следить за файлом и при добавлении в конец этого файла новых строк немедленно выводит их. Так продолжается до тех пор, пока пользователь не нажмет комбинацию клавиш CTRL+C.

tee — чтение со стандартного ввода и запись в стандартный вывод и в файлы

Linux предоставляет команду `tee`, которая создает Т-образное разветвление в конвейере. Программа `tee` читает данные со стандартного ввода и копирует их в стандартный вывод (чтобы дать возможность передать их дальше по конвейеру) и в один или несколько файлов. Это может пригодиться для сохранения промежуточных результатов обработки в конвейере. Ниже, продолжая один из

предыдущих примеров, мы сохраним полный список файлов в каталогах в файле *ls.txt*, перед тем как он будет отфильтрован командой `grep`:

```
[me@linuxbox ~]$ ls /usr/bin | tee ls.txt | grep zip
bunzip2
bzip2
gunzip
gzip
unzip
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
```

LINUX РАЗВИВАЕТ ВООБРАЖЕНИЕ

Когда меня просят объяснить разницу между Windows и Linux, я часто привожу аналогию с игрушками.

Windows — это как игровая приставка Game Boy. Вы идете в магазин и покупаете новенькую сияющую приставку с игрой в комплекте. Приносите ее домой, включаете и играете. Отличная графика, чудные звуки. Но спустя некоторое время игра надоедает. Вы опять идете в магазин и покупаете другую игру. Так повторяется снова и снова. Наконец, вы возвращаетесь в магазин и говорите человеку за прилавком: «Я хочу игру, которая делает это!» — а в ответ слышите, что такой игры не существует, потому что на нее нет спроса. Тогда вы говорите: «Но мне нужно всего лишь изменить вот это!» А продавец за прилавком говорит, что это невозможно. Игры продаются зашитыми в картриджи. И тут вы понимаете, что ваша приставка ограничена кругом игр, при создании которых кто-то другой решил за вас, что вам нужно, а что нет.

Linux, напротив, можно сравнить с самым большим в мире конструктором. Вы открываете коробку и видите необозримую коллекцию деталей — огромное число железных планочек, болтиков, гаечек, шестеренок, колесиков и моторчиков и несколько рекомендаций по сборке. Вы начинаете играть. Сначала создаете один предлагаемый образец, затем другой. Затем вы обнаруживаете, что у вас появились собственные идеи новых конструкций и механизмов. И вам не нужно возвращаться в магазин, потому что у вас уже есть все, что требуется. Конструктор формирует ваше воображение. Он позволяет создать то, что вы хотите.

Выбор игрушки, конечно же, дело глубоко личное, но признайтесь честно: какая игрушка принесла бы вам больше удовольствия?

Заключение

Как обычно, загляните в документацию с описанием каждой команды, представленной в этой главе. Здесь были показаны только самые общие примеры их использования, и все они имеют множество интересных параметров. По мере накопления опыта работы в Linux мы увидим, что поддержка перенаправления в командной оболочке чрезвычайно полезна для решения специализированных задач. Многие команды используют стандартный ввод и стандартный вывод, и почти все программы командной строки используют стандартный вывод ошибок для отображения информационных сообщений.

7 Взгляд на мир глазами командной оболочки

В этой главе мы посмотрим, что происходит в командной строке после нажатия клавиши ENTER. И в процессе исследования некоторых интересных и сложных механизмов командной оболочки будем пользоваться только одной новой командой:

echo — выводит строку текста.

Подстановка

Каждый раз, когда вы вводите команду и нажимаете ENTER, bash производит несколько операций с текстом, прежде чем выполнит вашу команду. Мы уже видели пару примеров, где простая последовательность символов, например *, может много значить для командной оболочки. Процесс, который происходит при этом, называется *подстановкой* (expansion). То есть вы вводите что-то, и это что-то замещается чем-то другим, прежде чем командная оболочка продолжит обработку. Чтобы показать, что все это значит, возьмем для примера команду echo — встроенную команду, выполняющую очень простую операцию: она выводит свои текстовые аргументы в стандартный поток вывода.

```
[me@linuxbox ~]$ echo this is a test
this is a test
```

Все очень просто. echo выведет любой свой аргумент. Давайте попробуем другой пример:

```
[me@linuxbox ~]$ echo *
Desktop Documents ls-output.txt Music Pictures Public Templates Videos
```

Что это? Почему echo не вывела символ *? Как вы помните из опытов с групповыми символами, символ * означает «последовательность любых символов в имени файла», правда, в том обзоре не рассказывалось, как командная оболочка делает это. На самом деле все просто: перед тем, как выполнить команду echo, оболочка замещает символ * чем-то другим (в данном случае именами файлов в текущем рабочем каталоге). После нажатия клавиши ENTER командная оболочка автоматически производит подстановку любых групповых символов в командной строке, прежде чем выполнить ее, поэтому команда echo не увидела * — она получила уже готовый результат после подстановки. Теперь вы понимаете, что в действительности echo действует в точности с нашими ожиданиями?

Подстановка путей

Механизм работы групповых символов называется *подстановкой пути* (pathname expansion). Если вернуться к некоторым приемам, продемонстрированным в предыдущих главах, мы увидим, что в действительности они основаны на подстановке. Допустим, содержимое домашнего каталога выглядит вот так:

```
[me@linuxbox ~]$ ls
Desktop    ls-output.txt  Pictures  Templates
Documents  Music          Public    Videos
```

Мы могли бы выполнить следующую подстановку:

```
[me@linuxbox ~]$ echo D*
Desktop Documents
```

или

```
[me@linuxbox ~]$ echo *s
Documents Pictures Templates Videos
```

или даже

```
[me@linuxbox ~]$ echo [[:upper:]]*
Desktop Documents Music Pictures Public Templates Videos
```

И заглянуть за пределы домашнего каталога:

```
[me@linuxbox ~]$ echo /usr/*/share
/usr/kerberos/share /usr/local/share
```

ПОДСТАНОВКА ПУТИ ДЛЯ СКРЫТЫХ ФАЙЛОВ

Как мы знаем, файлы с именами, начинающимися с точки, считаются скрытыми. Механизм подстановки пути также учитывает это. Подстановка, такая как

```
echo *
```

не покажет скрытые файлы.

На первый взгляд кажется, что можно было бы включить скрытые файлы в подстановку, добавив в начало шаблона точку, например:

```
echo .*
```

Да, такой подход даст желаемое. Однако, если внимательно исследовать результаты, можно заметить, что в них также присутствуют имена `.` (точка) и `..` (две точки). Так как эти имена соответствуют текущему рабочему каталогу и родительскому каталогу, применение такого шаблона может привести к неправильным результатам. Убедимся в этом с помощью команды

```
ls -d .* | less
```

Чтобы обеспечить правильную подстановку пути в такой ситуации, следует использовать специализированный шаблон. Следующий шаблон действует правильно:

```
ls -d .[!.]?*
```

Этот шаблон замещается именами файлов, начинающимися с единственной точки, за которой следует любое количество других символов. Он будет работать правильно с большинством имен скрытых файлов (кроме тех, что начинаются с нескольких точек). Команда `ls` с параметром `-A` («almost all» — «почти все») выведет список всех скрытых файлов без ошибок.

```
ls -A
```

Подстановка тильды

Как вы помните из вводного пояснения команды `cd`, символ «тильда» (`~`) имеет специальное значение. Если он используется в начале слова, то замещается именем домашнего каталога указанного пользователя или, если пользователь не указан, именем домашнего каталога текущего пользователя:

```
[me@linuxbox ~]$ echo ~
/home/me
```

Если в системе существует учетная запись пользователя foo, тогда

```
[me@linuxbox ~]$ echo ~foo
/home/foo
```

Подстановка результатов арифметических выражений

Командная оболочка поддерживает также подстановку результатов арифметических выражений. Это позволяет использовать командную строку как калькулятор:

```
[me@linuxbox ~]$ echo $((2 + 2))
4
```

Для подстановки арифметических выражений используется следующий формат:

```
$((выражение))
```

где *выражение* — это арифметическое выражение, состоящее из значений и арифметических операторов.

Механизм подстановки арифметических выражений позволяет использовать только целые числа (невещественные), зато поддерживает множество арифметических операций. В табл. 7.1 перечислены некоторые из поддерживаемых операторов.

Таблица 7.1. Арифметические операторы

Оператор	Описание
+	Сложение
-	Вычитание
*	Умножение
/	Деление (но помните, из-за того, что подстановка поддерживает только целочисленную арифметику, результатом будет целое число)
%	Деление по модулю или остаток от деления
**	Возведение в степень

Пробелы в арифметических выражениях не играют роли, а выражения могут содержать вложенные выражения. Например, умножение 5^2 на 3:

```
[me@linuxbox ~]$ echo $(($((5**2)) * 3))
75
```

Для группировки подвыражений допускается использование одиночных круглых скобок. С помощью этого приема выражение, приведенное выше, можно

переписать, как показано ниже, и получить тот же результат, но при этом будет использоваться одна операция подстановки вместо двух:

```
[me@linuxbox ~]$ echo $(((5**2) * 3))
75
```

Следующий пример демонстрирует использование операторов деления и получения остатка. Обратите внимание, как действует целочисленное деление:

```
[me@linuxbox ~]$ echo Пять разделить на два будет $((5/2))
Пять разделить на два будет 2
[me@linuxbox ~]$ echo и $((5%2)) в остатке.
и 1 в остатке.
```

Подстановка результатов арифметических выражений подробнее будет рассматриваться в главе 34.

Подстановка фигурных скобок

Самым малопонятным, пожалуй, выглядит результат подстановки фигурных скобок. С помощью этого механизма из одного шаблона, содержащего фигурные скобки, создается множество текстовых строк. Например:

```
[me@linuxbox ~]$ echo Впереди-{A,B,C}-позади
Впереди-A-позади Впереди-B-позади Впереди-C-позади
```

Шаблоны с фигурными скобками могут содержать начальную часть, которая называется *преамбулой*, и заключительную часть, которая называется *эпилогом*. Внутри фигурных скобок находится список строк, разделенных запятыми, или диапазон целых чисел или одиночных символов. Использование пробелов внутри фигурных скобок не допускается. Ниже приводится пример с использованием диапазона целых чисел:

```
[me@linuxbox ~]$ echo Число_{1..5}
Число_1 Число_2 Число_3 Число_4 Число_5
```

В версии bash 4.0 и выше целые числа можно *дополнять ведущими нулями*, например:

```
[me@linuxbox ~]$ echo {01..15}
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15
[me@linuxbox ~]$ echo {001..15}
001 002 003 004 005 006 007 008 009 010 011 012 013 014 015
```

В следующем примере используется диапазон символов в обратном порядке:

```
[me@linuxbox ~]$ echo {Z..A}
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
```

Допускается вложение фигурных скобок:

```
[me@linuxbox ~]$ echo a{A{1,2},B{3,4}}b
aA1b aA2b aB3b aB4b
```

Какую пользу можно извлечь из этого? Такая возможность может пригодиться для формирования списков файлов или каталогов, которые требуется создать. Например, фотограф, имеющий огромную коллекцию фотографий и желающий организовать ее по годам и месяцам, мог бы начать с создания группы каталогов с именами, состоящими из номера года и месяца. Благодаря этому имена каталогов будут отсортированы в хронологическом порядке. Можно было бы ввести полный список каталогов, но это обременительно и чревато ошибками. Вместо этого выполним следующую команду:

```
[me@linuxbox ~]$ mkdir Photos
[me@linuxbox ~]$ cd Photos
[me@linuxbox Photos]$ mkdir {2007..2009}-{01..12}
[me@linuxbox Photos]$ ls
2007-01  2007-07  2008-01  2008-07  2009-01  2009-07
2007-02  2007-08  2008-02  2008-08  2009-02  2009-08
2007-03  2007-09  2008-03  2008-09  2009-03  2009-09
2007-04  2007-10  2008-04  2008-10  2009-04  2009-10
2007-05  2007-11  2008-05  2008-11  2009-05  2009-11
2007-06  2007-12  2008-06  2008-12  2009-06  2009-12
```

Однако!

Подстановка параметров

В этой главе мы лишь кратко коснемся подстановки параметров, а детально обсудим ее позже. Эта возможность полезнее в сценариях на языке командной оболочки, чем непосредственно в командной строке. Многие из ее возможностей имеют отношение к способности системы хранить маленькие фрагменты данных и присваивать этим фрагментам имена. Многие такие фрагменты, правильнее их называть *переменными*, уже существуют и доступны для исследования. Например, переменная с именем `USER` хранит ваше имя пользователя. Подстановка параметра и получение содержимого переменной `USER` выполняется следующим образом:

```
[me@linuxbox ~]$ echo $USER
me
```

Чтобы увидеть список доступных переменных, выполните следующую команду:

```
[me@linuxbox ~]$ printenv | less
```

Возможно, вы обратили внимание, что если в других вариантах подстановки допустить ошибку в шаблоне, подстановка не будет выполнена и команда `echo` просто выведет ошибочный шаблон. В случае с подстановкой параметров все иначе: если ошибиться в имени переменной, подстановка все равно будет выполнена, но результатом будет пустая строка:

```
[me@linuxbox ~]$ echo $SUER
[me@linuxbox ~]$
```

Подстановка команд

Подстановка команд позволяет использовать поток вывода команд в качестве аргументов других команд:

```
[me@linuxbox ~]$ echo $(ls)
Desktop Documents ls-output.txt Music Pictures Public Templates Videos
```

Один из моих любимых вариантов выглядит так:

```
[me@linuxbox ~]$ ls -l $(which cp)
-rwxr-xr-x 1 root root 71516 2007-12-05 08:58 /bin/cp
```

Здесь результат команды `which cp` передается как аргумент команде `ls`, благодаря чему мы получаем информацию о программе `cp`, не зная полного пути к ней. Подстановка команд не ограничивается такими простыми командами. Можно использовать целые конвейеры (здесь показана только часть вывода):

```
[me@linuxbox ~]$ file $(ls -d /usr/bin/* | grep zip)
/usr/bin/bunzip2:      symbolic link to `bzip2'
/usr/bin/bzip2:        ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV ),
dynamically linked (uses shared libs), for GNU/Linux 2.6.9, stripped
/usr/bin/bzip2recover: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.9, stripped
/usr/bin/funzip:       ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.9, stripped
/usr/bin/gpg-zip:      Bourne shell script text executable
/usr/bin/gunzip:       symbolic link to `../../bin/gunzip'
/usr/bin/gzip:         symbolic link to `../../bin/gzip'
/usr/bin/mzip:         symbolic link to `mtools'
```

В этом примере результаты конвейера превратились в список аргументов команды `file`.

Механизм подстановки команд имеет альтернативный синтаксис, унаследованный от более старых командных оболочек, который также поддерживается в bash. В нем вместо знака доллара и круглых скобок используются *обратные апострофы*:

```
[me@linuxbox ~]$ ls -l `which cp`
-rwxr-xr-x 1 root root 71516 2012-12-05 08:58 /bin/cp
```

Экранирование

Теперь, после знакомства с множеством способов подстановки, поддерживаемых командной оболочкой, можно начинать учиться управлять ими. Например, взгляните на эту команду:

```
[me@linuxbox ~]$ echo this is a     test
this is a test
```

Или на эту:

```
[me@linuxbox ~]$ echo Итого $100.00
Итого 00.00
```

В первом примере механизм *разбиения на слова* удалил дополнительные пробелы из списка аргументов команды echo. Во втором — механизм подстановки параметров подставил пустую строку вместо $1, потому что не нашел такую переменную. Командная оболочка предоставляет механизм, который называется *экранированием* (quoting), для выборочного подавления нежелательной подстановки.

Двойные кавычки

Первый тип экранирования, который мы рассмотрим, — *двойные кавычки*. Если заключить текст в двойные кавычки, все специальные символы потеряют свое специальное значение и будут интерпретироваться как обычные символы. Исключение составляют: $ (знак доллара), \ (обратный слеш) и ` (обратный апостроф). То есть разбиение на слова, подстановка путей, подстановка тильды и подстановка фигурных скобок выполняться не будут, но подстановка параметров, подстановка значений арифметических выражений и подстановка команд все еще будут выполняться. Благодаря двойным кавычкам мы сможем обрабатывать имена файлов с пробелами. Представьте, что мы по ошибке создали файл с именем Два слова.txt. Если попытаться использовать это имя в командной строке, механизм разбиения слов будет интерпретировать его как два отдельных аргумента:

```
[me@linuxbox ~]$ ls -l Два слова.txt
ls: невозможно получить доступ к 'Два': Нет такого файла или каталога
ls: невозможно получить доступ к 'слова.txt': Нет такого файла или каталога
```

Добавив двойные кавычки, можно запретить разбиение слов и получить желаемый результат; кроме того, с помощью двойных кавычек мы исправим ошибку:

```
[me@linuxbox ~]$ ls -l "Два слова.txt"
-rw-rw-r-- 1 me me 18 2012-02-20 13:03 Два слова.txt
[me@linuxbox ~]$ mv "Два слова.txt" Два_слова.txt
```

Вот так! Теперь не нужно вводить эти противные двойные кавычки.

Запомните: подстановка параметров, подстановка значений арифметических выражений и подстановка команд все еще выполняются в двойных кавычках:

```
[me@linuxbox ~]$ echo "$USER $((2+2)) $(cal)"
me 4    February 2020
Su Mo Tu We Th Fr Sa
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
```

Давайте отвлечемся и посмотрим, какой эффект оказывают двойные кавычки на подстановку команд. Сначала рассмотрим действие механизма разбиения на слова. В одном из примеров, приведенных выше, мы видели, как механизм разбиения на слова удаляет дополнительные пробелы из текста:

```
[me@linuxbox ~]$ echo this is a     test
this is a test
```

По умолчанию этот механизм находит пробелы, символы табуляции и символы перевода строки и интерпретирует их как *разделители* слов. То есть вне кавычек упомянутые символы не считаются частью текста. Они являются лишь разделителями. Поскольку они делят слова на аргументы, получается, что в нашем примере командная строка состоит из команды и четырех аргументов. Однако если добавить двойные кавычки, разбиение на слова выполняться не будет и внутренние пробелы не будут считаться разделителями — они станут частью аргумента:

```
[me@linuxbox ~]$ echo "this is a     test"
this is a     test
```

После добавления двойных кавычек командная строка будет состоять из команды и одного аргумента.

Тот факт, что символы перевода строки интерпретируются механизмом разбиения на слова как разделители, вызывает интересный и трудноуловимый эффект при подстановке команд. Взгляните:

```
[me@linuxbox ~]$ echo $(cal)
February 2020 Su Mo Tu We Th Fr Sa 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
[me@linuxbox ~]$ echo "$(cal)"
   February 2020
Su Mo Tu We Th Fr Sa
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
```

В первом случае подстановка команд без кавычек привела к созданию командной строки с 38 аргументами, а во втором случае получилась командная строка с одним аргументом, включающим внутренние пробелы и символы перевода строки.

Одиночные кавычки

Если вам требуется подавить *все* подстановки, используйте *одиночные кавычки*. Ниже для сравнения приводятся результаты неэкранированной команды и команды, экранированной двойными и одиночными кавычками:

```
[me@linuxbox ~]$ echo text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
text /home/me/ls-output.txt a b foo 4 me
[me@linuxbox ~]$ echo "text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER"
text ~/*.txt {a,b} foo 4 me
[me@linuxbox ~]$ echo 'text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER'
text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
```

Как видите, каждый следующий уровень экранирования все больше и больше подавляет подстановку.

Экранирование символов

Иногда бывает необходимо экранировать только один символ. Для этого достаточно добавить перед символом обратный слеш, который в данном случае называется *экранирующим символом* (escape character). Часто этот прием используется в двойных кавычках, чтобы выборочно предотвратить подстановку.

```
[me@linuxbox ~]$ echo "Баланс счета пользователя $USER: \$5.00"
Баланс счета пользователя me: $5.00
```

Экранирование символов также широко применяется для подавления специального значения символов в именах файлов. Например, в именах файлов допускается использование символов, которые имеют специальное значение для командной оболочки. К их числу относятся: `$`, `!`, `&`, пробел и др. Чтобы включить специальный символ в имя файла, его достаточно экранировать, как показано ниже:

```
[me@linuxbox ~]$ mv bad\&filename good_filename
```

Чтобы включить сам экранирующий символ, его также нужно экранировать, введя `\\`. Имейте в виду, что внутри одиночных кавычек обратный слеш теряет свое специальное значение и интерпретируется как обычный символ.

Управляющие последовательности

Обратный слеш используется не только в роли экранирующего символа, но и как часть специальных символов, которые называют *управляющими кодами* (control codes). Первые 32 символа в схеме кодирования ASCII использовались для передачи различных команд в устройствах, таких как телетайп. Некоторые из этих кодов хорошо знакомы вам (табуляция, забой, перевод строки и возврат каретки), тогда как другие — нет (пустой символ, конец передачи и подтверждение).

В табл. 7.2 перечислены некоторые наиболее известные управляющие последовательности.

Таблица 7.2. Управляющие последовательности

Управляющая последовательность	Значение
`\a`	Звонок («предупреждение» — заставляет компьютер подать звуковой сигнал)
`\b`	Забой (backspace)
`\n`	Новая строка (в Unix-подобных системах этот символ выполняет перевод строки)
`\r`	Возврат каретки
`\t`	Табуляция

Идея использования обратного слеша зародилась в языке программирования С и была заимствована многими другими языками, включая язык командной оболочки.

Параметр `-e` команды `echo` включает интерпретацию управляющих последовательностей. Их можно также заключать в конструкцию `$' '`. Ниже демонстрируется использование команды `sleep` для создания элементарного таймера, простой программы, которая всего лишь ждет указанное число секунд и завершается.

```
sleep 10; echo -e "Time's up\a"
```

То же самое можно выразить так:

```
sleep 10; echo "Time's up" $'\a'
```

Заключение

По мере накопления опыта использования командной оболочки мы все чаще будем использовать возможности подстановки и экранирования, поэтому важно хорошо понимать, как они работают. Фактически можно смело утверждать, что эти два механизма являются наиболее важными для изучения аспектами командной оболочки. Без надлежащего понимания того, как действует подстановка, командная оболочка будет оставаться источником непонимания и домыслов, при этом многие ее возможности останутся неиспользованными.

8 Продвинутые приемы работы с клавиатурой

Я часто шутливо описываю Unix как «операционную систему для тех, кто любит печатать». Казалось бы, сам факт наличия командной строки доказывает это. Но в действительности пользователи командной строки не любят печатать *слишком* много. Зачем, если есть так много команд с короткими именами, таких как `cp`, `ls`, `mv` и `rm`?

Фактически одной из самых заветных целей командной строки является уменьшение объема ввода — возможность выполнить большую часть работы всего несколькими нажатиями клавиш. Другая цель — не позволить рукам оторваться от клавиатуры и коснуться мыши. В этой главе мы рассмотрим возможности `bash`, увеличивающие скорость и эффективность использования клавиатуры.

Здесь будут представлены следующие команды:

`clear` — очищает экран;

`history` — выводит содержимое истории команд.

Редактирование командной строки

Для поддержки операций редактирования командной строки `bash` использует библиотеку (коллекцию подпрограмм, которую могут использовать разные программы) с именем *Readline*. Мы уже видели некоторые из них. Например, нам знакомы клавиши со стрелками влево и вправо, перемещающие курсор, но существует еще целое множество других операций. Рассматривайте их как дополнительные

инструменты, которые можно использовать в работе. Необязательно стремиться изучить их все, но многие из них весьма практичны. Выбирайте те, что вам понравятся.

ПРИМЕЧАНИЕ

Некоторые комбинации клавиш, описываемые далее (особенно те, что включают клавишу ALT), могут перехватываться графическим интерфейсом и использоваться для выполнения других функций. Однако все комбинации без исключения должны правильно работать в виртуальной консоли.

Перемещение курсора

В табл. 8.1 перечислены комбинации клавиш, используемые для перемещения курсора.

Таблица 8.1. Команды перемещения курсора

Клавиша	Действие
CTRL+A	Перемещает курсор в начало строки
CTRL+E	Перемещает курсор в конец строки
CTRL+F	Перемещает курсор на один символ вперед; действует так же, как клавиша со стрелкой вправо
CTRL+B	Перемещает курсор на один символ назад; действует так же, как клавиша со стрелкой влево
ALT+F	Перемещает курсор на одно слово вперед
ALT+B	Перемещает курсор на одно слово назад
CTRL+L	Очищает экран и устанавливает курсор в левый верхний угол. То же самое делает команда `clear`

Изменение текста

Вводя команды, легко допустить ошибку, поэтому нам нужен способ, позволяющий быстро исправлять их. В табл. 8.2 перечислены комбинации клавиш для редактирования символов в командной строке.

КЛАВИША META

Отважившиеся заглянуть в документацию к Readline, которая находится в разделе «READLINE», на странице справочного руководства (man) для `bash`, столкнутся с термином *клавиша meta* (meta key). На современных клавиатурах ей соответствует клавиша ALT, но так было не всегда.

В стародавние времена (до появления IBM-совместимых персональных компьютеров, но после появления Unix) персональные компьютеры не были так широко распространены. Иногда их заменяли устройства, называемые *терминалами*. Терминал — это коммуникационное устройство с текстовым дисплеем и клавиатурой, имеющее внутри столько электроники, сколько необходимо для отображения символов и перемещения курсора. Терминалы подключались (обычно посредством последовательного кабеля) к большому компьютеру или коммуникационной сети большого компьютера. В то время существовало очень много различных терминалов, имевших разные клавиатуры и дисплеи с разными функциональными возможностями. Так как все они поддерживали как минимум набор символов ASCII, разработчикам программного обеспечения, пишущим переносимые приложения, необходимо было прийти к общему знаменателю. В системах Unix применяется очень сложный способ использования терминалов и их разнообразных возможностей. Поскольку разработчики Readline не были уверены в наличии специализированной управляющей клавиши, они изобрели ее и назвали *meta*. На современных клавиатурах роль клавиши meta играет ALT, однако если вы все еще используете терминал (до сих пор поддерживаются в Linux!), можно просто нажать и отпустить клавишу ESC, и вы получите эффект нажатия и удержания клавиши ALT.

Таблица 8.2. Команды редактирования текста

Клавиша	Действие
CTRL+D	Удаляет символ в позиции курсора
CTRL+T	Меняет местами два символа — в позиции курсора и предшествующий ему
ALT+T	Меняет местами два слова — в позиции курсора и предшествующее ему
ALT+L	Переводит в нижний регистр символы, начиная с символа в позиции курсора и до конца слова
ALT+U	Переводит в верхний регистр символы, начиная с символа в позиции курсора и до конца слова

Вырезание и вставка (удаление и возврат) текста

В документации к Readline используется термин *killing and yanking* (удаление и возврат), обозначающий операцию, которую обычно называют вырезанием и вставкой (cutting and pasting). В табл. 8.3 перечислены комбинации клавиш, выполняющие вырезание и вставку. Вырезанные элементы сохраняются в кольцевом буфере, который называется *kill-ring* (кольцо удалений).

Таблица 8.3. Команды вырезания и вставки

Клавиша	Действие
CTRL+K	Удаляет символы от позиции курсора до конца строки
CTRL+U	Удаляет символы от позиции курсора до начала строки
ALT+D	Удаляет символы от позиции курсора до конца текущего слова
ALT+BACKSPACE	Удаляет символы от позиции курсора до начала текущего слова. Если курсор находится в начале слова, удаляется предшествующее слово
CTRL+Y	Извлекает текст из кольцевого буфера удалений и вставляет его в позицию курсора

Дополнение

Другой вариант помощи пользователям реализован в командной оболочке в виде механизма *дополнения* (completion). Дополнение происходит, когда в процессе ввода команды нажимается клавиша TAB. Давайте посмотрим, как это работает. Допустим, что ваш домашний каталог содержит следующее:

```
 [me@linuxbox ~]$ ls
Desktop     ls-output.txt  Pictures  Templates  Videos
Documents   Music          Public
```

Попробуйте ввести следующую строку, но *не нажимайте* клавишу ENTER:

```
[me@linuxbox ~]$ ls l
```

Теперь нажмите клавишу TAB:

```
[me@linuxbox ~]$ ls ls-output.txt
```

Обратили ли вы внимание, как командная оболочка дополнила командную строку за вас? Попробуйте теперь набрать следующую строку — и снова не нажимайте ENTER):

```
[me@linuxbox ~]$ ls D
```

Нажмите TAB:

```
[me@linuxbox ~]$ ls D
```

Дополнения не произошло — просто прозвучал звуковой сигнал. Так получилось потому, что символу `D` соответствует более одного элемента в каталоге. Чтобы командная оболочка дополнила вашу строку, предложенная вами «подсказка» должна иметь однозначное продолжение. Попробуйте продолжить ввод:

```
[me@linuxbox ~]$ ls Do
```

Затем нажмите TAB:

```
[me@linuxbox ~]$ ls Documents
```

Дополнение произошло.

Этот пример демонстрирует дополнение путей как наиболее частый случай использования дополнения. Однако дополнение также работает с именами переменных (когда слово начинается с символа `$`), именами пользователей (когда слово начинается с символа `~`), командами (когда дополняемое слово является первым в командной строке) и сетевыми именами компьютеров (когда слово начинается с символа `@`). Дополнение сетевых имен компьютеров действует только в отношении имен, перечисленных в */etc/hosts*.

С механизмом дополнения связано несколько управляющих комбинаций клавиш (табл. 8.4).

Таблица 8.4. Команды дополнения

Клавиша	Действие
ALT+?	Выводит список возможных дополнений. В большинстве систем тот же эффект можно получить, нажав клавишу TAB второй раз, что намного проще
ALT+*	Вставит все возможные дополнения. Это пригодится в том случае, если требуется использовать больше одного возможного варианта дополнения

Существует еще несколько команд, смысл которых для меня не совсем ясен. Полный список вы сможете найти на странице справочного руководства (man) для `bash`, в разделе «READLINE».

> **ПРОГРАММИРУЕМОЕ ДОПОЛНЕНИЕ**
>
> Последние версии bash реализуют механизм *программируемого дополнения*. Программируемое дополнение дает возможность добавлять дополнительные правила. Обычно это делается с целью добавить поддержку определенных приложений. Например, можно добавить дополнение списка параметров команды или файлов определенного типа, поддерживаемых приложением. В Ubuntu определено огромное множество таких правил. Программируемое дополнение реализуется посредством функций командной оболочки — небольших мини-сценариев, о которых будет рассказываться в следующих главах. Если вам любопытно, попробуйте выполнить команду
>
> ```
> set | less
> ```
>
> и вы увидите их. Однако не все дистрибутивы включают эти функции по умолчанию.

Использование истории

Как рассказывалось в главе 1, bash поддерживает историю вводившихся команд. Этот список команд хранится в домашнем каталоге, в файле с именем *.bash_history*. Механизм истории помогает уменьшить объем ручного ввода, особенно в сочетании с командами редактирования командной строки.

Поиск в истории

Просмотреть содержимое истории можно в любой момент с помощью команды:

```
[me@linuxbox ~]$ history | less
```

По умолчанию bash хранит последние 500 введенных команд, однако в большинстве современных дистрибутивов этот предел увеличен до 1000. Как изменить это значение, мы узнаем в главе 11. А теперь представим, что вам понадобилось найти команды, использовавшиеся для получения списка содержимого */usr/bin*. Вот один из возможных способов:

```
[me@linuxbox ~]$ history | grep /usr/bin
```

А теперь представим, что среди результатов нужно выбрать запись с интересующей вас командой:

```
88  ls -l /usr/bin > ls-output.txt
```

Здесь число 88 — это порядковый номер записи команды в списке истории. Зная это число, можно воспользоваться еще одной разновидностью подстановки, которая называется *подстановкой записей истории* (history expansion). Для этого введите:

```
[me@linuxbox ~]$ !88
```

и bash заменит !88 содержимым 88-й записи в списке истории. Подробнее об этой форме подстановки записей истории мы поговорим чуть ниже.

bash также дает возможность выполнять поступательный поиск в списке истории. Это означает, что bash может выполнять поиск в списке истории по мере ввода символов, уточняя результаты с вводом каждого нового символа. Чтобы запустить поступательный поиск, нажмите комбинацию CTRL+R и введите искомый текст. Закончив поиск, нажмите ENTER, чтобы выполнить команду, или CTRL+J, чтобы скопировать запись из списка истории в текущую командную строку. Чтобы найти следующее вхождение текста (переместиться «вверх» по списку истории), нажмите CTRL+R еще раз. Чтобы завершить поиск, нажмите CTRL+G или CTRL+C. Следующий пример демонстрирует, как действует поиск:

```
[me@linuxbox ~]$
```

Первое нажатие комбинации CTRL+R:

```
(reverse-i-search)`':
```

Приглашение к вводу изменится, показывая, что выполняется поступательный поиск в обратном порядке. Под словами «в обратном порядке» подразумевается, что поиск выполняется от «текущего момента» до некоторого момента в прошлом. Далее мы начинаем ввод искомого текста, в данном примере */usr/bin*:

```
(reverse-i-search)`/usr/bin': ls -l /usr/bin > ls-output.txt
```

Механизм поиска сразу же возвращает результат. Теперь, чтобы выполнить найденную команду, необходимо нажать ENTER, или вы можете скопировать команду в командную строку для дальнейшего редактирования, нажав CTRL+J. Давайте скопируем ее. Нажмите CTRL+J:

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Механизм поиска вернет управление, командная строка заполнится и будет готова для выполнения!

В табл. 8.5 перечислены некоторые комбинации клавиш, используемые для манипуляций со списком истории команд.

Таблица 8.5. Команды для работы с историей

Клавиша	Действие
CTRL+P	Переход к предыдущей записи в истории. Действует так же, как клавиша со стрелкой вверх
CTRL+N	Переход к следующей записи в истории. Действует так же, как клавиша со стрелкой вниз
ALT+<	Переход в начало (к первой записи) списка истории
ALT+>	Переход в конец (к последней записи) списка истории
CTRL+R	Инкрементальный поиск в обратном порядке. Поиск выполняется поступательно, от текущей записи вверх по списку истории
ALT+P	Поиск в обратном порядке, не инкрементальный. При использовании этого вида поиска введите искомую строку и нажмите ENTER, и только после этого будет выполнен фактический поиск
ALT+N	Поиск в прямом порядке, не поступательный
CTRL+O	Выполните текущую команду в списке истории и перейдите к следующей. Эту комбинацию удобно использовать, если требуется повторно выполнить последовательность команд из списка истории

Подстановка записей истории

Командная оболочка поддерживает специализированный вид подстановки — подстановку записей из списка истории при использовании символа `!`. Мы уже видели, как восклицательный знак, сопровождаемый числом, замещается записью из списка истории. Этот вид подстановки имеет несколько разновидностей (табл. 8.6).

Таблица 8.6. Команды механизма подстановки записей истории

Последовательность	Действие
`!!`	Повторяет последнюю команду. Проще, пожалуй, нажать клавишу со стрелкой вверх и ENTER
`!`*число*	Повторяет команду из записи с указанным номером
`!`*строка*	Повторяет последнюю команду в списке истории, начинающуюся с указанной строки
`!?`*строка*	Повторяет последнюю команду в списке истории, содержащую указанную строку

Не используйте формы `!строка` и `!?строка`, если только вы абсолютно точно не знаете содержимого записей в списке истории.

Механизм подстановки записей истории поддерживает также другие комбинации, но эта тема становится слишком запутанной, и мы не станем перегружать себя лишней информацией. Желающие смогут обратиться к странице справочного руководства (man) для `bash`, в разделе «HISTORY EXPANSION». Загляните туда!

SCRIPT

В дополнение к истории команд в `bash` большинство дистрибутивов Linux включают программу `script`, которую можно использовать для записи в файлы целых сеансов работы с командной оболочкой. Базовый синтаксис команды:

```
script [файл]
```

где `файл` — это имя файла для записи. Если файл не будет указан, сохранение сеанса будет произведено в файл `typescript`. Полное описание параметров и возможностей программы можно найти на странице справочного руководства (man) для `script`.

Заключение

В этой главе мы рассмотрели несколько приемов работы с клавиатурой, поддерживаемых командной оболочкой, с целью помочь истинным фанатам клавиатуры уменьшить объем работы. Я думаю, что потом, когда вы сроднитесь с командной строкой, вы сможете обратиться к этой главе, чтобы вспомнить описанные здесь приемы. А пока будем считать их необязательными, но потенциально полезными.

9 Привилегии

Операционные системы, следующие традициям Unix, отличаются от систем, следующих традициям MS-DOS, тем, что являются не только многозадачными, но и многопользовательскими.

Что это означает на самом деле? Это означает, что компьютером могут одновременно пользоваться несколько человек. Несмотря на то что обычно компьютер имеет всего одну клавиатуру и монитор, это обстоятельство не мешает совместному пользованию. Например, если компьютер подключен к локальной сети или к Интернету, удаленные пользователи смогут зайти на него через `ssh` (secure shell — безопасная командная оболочка) и выполнять операции. Фактически удаленные пользователи могут запускать приложения с графическим интерфейсом и получать изображение на удаленном дисплее. X Window System поддерживает такую возможность изначально.

Поддержка многопользовательского режима работы — не недавнее «изобретение» Linux, а возможность, глубоко внедренная в архитектуру операционной системы. Учитывая окружение, в котором создавалась система Unix, это имело определенный смысл. В те времена, когда компьютеры еще не были «персональными», они были большими и дорогими. Типичная компьютерная система университета, например, состояла из большого центрального компьютера в одном здании и терминалов, разбросанных по всему университетскому городку и соединенных с большим центральным компьютером. Компьютер мог одновременно обслуживать множество пользователей.

Чтобы подобная возможность имела практическую ценность, необходим способ определенной «изоляции» пользователей друг от друга. В конце концов, действия рядового пользователя не должны приводить к аварийному завершению работы компьютера, и ни один пользователь не должен иметь возможность вносить изменения в файлы, принадлежащие другому пользователю.

В данной главе мы рассмотрим эту важную сторону безопасности системы и познакомимся со следующими командами:

`id` — выводит информацию об идентичности пользователя;

`chmod` — изменяет режим доступа к файлу;

`umask` — определяет разрешения доступа к файлам по умолчанию;

`su` — запускает командную оболочку от имени другого пользователя;

`sudo` — выполняет команду от имени другого пользователя;

`chown` — изменяет владельца файла;

`chgrp` — изменяет группу файла;

`passwd` — изменяет пароль пользователя.

Владельцы, члены группы и все остальные

Знакомясь с системой в главе 4, вы уже сталкивались со следующей проблемой при исследовании файлов, таких как */etc/shadow*:

```
[me@linuxbox ~]$ file /etc/shadow
/etc/shadow: Обычный файл, нет прав на чтение
[me@linuxbox ~]$ less /etc/shadow
/etc/shadow: Отказано в доступе
```

Причина этого сообщения об ошибке заключается в том, что обычные пользователи не имеют права читать этот файл.

В модели безопасности Unix пользователь может *владеть* файлами и каталогами. Если пользователь владеет файлом или каталогом, он может управлять доступом к нему. Пользователи могут также принадлежать *группе*, состоящей из одного или нескольких пользователей, и получить права доступа к файлам и каталогам для членов группы, которые определяются владельцами. Кроме прав доступа для группы, владелец может также определить некоторые права доступа для всех остальных, их в терминологии Unix называют *мир* (world). Получить информацию о своей идентичности можно с помощью команды `id`:

```
[me@linuxbox ~]$ id
uid=500(me) gid=500(me) groups=500(me)
```

Давайте рассмотрим этот вывод. Когда создается учетная запись пользователя, ей присваивается число, которое называют *идентификатором пользователя* (user ID), или *uid*. Это число, исключительно ради удобства человека, отображается как имя пользователя. Пользователю назначается *идентификатор основной группы* (primary group ID), или *gid*, и дополнительно пользователь может включаться в состав других групп. Предыдущий пример взят из системы Fedora. В других системах, таких как Ubuntu, вывод команды может немного отличаться.

```
 [me@linuxbox ~]$ id
uid=1000(me) gid=1000(me) groups=4(adm),20(dialout),24(cdrom),25(floppy),29(audio),
30(dip),44(video),
46(plugdev),108(lpadmin),114(admin),1000(me)
```

Как видите, в Ubuntu числа uid и gid отличаются. Это объясняется тем, что в Fedora нумерация учетных записей обычных пользователей начинается с 500, тогда как в Ubuntu — с 1000. Кроме того, пользователь в Ubuntu принадлежит множеству других групп. Это связано с особенностями управления привилегиями доступа к системным устройствам и службам в Ubuntu.

А где же вся эта информация хранится? Как и многое другое в Linux, она хранится в паре текстовых файлов. Учетные записи пользователей хранятся в файле */etc/passwd*, а информация о группах — в файле */etc/group*. Когда создаются новые учетные записи и группы, эти файлы изменяются вместе с файлом */etc/shadow*, где хранится информация о пароле пользователя. Для каждой учетной записи в файле */etc/passwd* определяется имя пользователя (для входа), числовой идентификатор пользователя (uid), числовой идентификатор основной группы (gid), действительное имя пользователя, путь к домашнему каталогу и командная оболочка входа (login shell). Заглянув внутрь */etc/passwd* и */etc/group*, можно заметить, что помимо учетных записей обычных пользователей здесь также хранятся учетные записи суперпользователя (uid 0) и различных других системных пользователей.

В следующей главе, где рассказывается о процессах, вы узнаете, что некоторые из этих других «пользователей» в действительности существуют не просто так.

Несмотря на то что во многих Unix-подобных системах обычных пользователей включают в общую группу, такую как `users`, в современных дистрибутивах Linux принято создавать для каждого пользователя свою, уникальную группу с одним членом и именем, совпадающим с именем пользователя. Это упрощает распределение определенных типов привилегий.

Чтение, запись и выполнение

Права доступа к файлам и каталогам определяются в терминах права на чтение, права на запись и права на выполнение. Если взглянуть на вывод команды `ls`, можно увидеть некоторые подсказки о том, как эти права реализованы:

```
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me    me   0 2018-03-06 14:52 foo.txt
```

Первые 10 символов в выводе — это *атрибуты файла*. Первый из этих символов определяет *тип файла*. В табл. 9.1 перечислены типы файлов, которые чаще всего встречаются на практике (существуют также другие, реже используемые типы файлов).

Таблица 9.1. Типы файлов

Атрибут	Тип файла
-	Обычный файл
d	Каталог
l	Символическая ссылка. Обратите внимание, что для символических ссылок все остальные атрибуты имеют значение `rwxrwxrwx` и не отражают действительные права доступа. Фактические права доступа к файлу определяются атрибутами самого файла, на который указывает символическая ссылка
c	*Специальный файл символьного устройства*. Файлы этого типа соответствуют устройствам, таким как терминал или модем, которые обрабатывают данные как потоки байтов
b	*Специальный файл блочного устройства*. Файлы этого типа соответствуют устройствам, таким как привод жесткого диска или CD-ROM, которые обрабатывают данные блоками

Остальные девять символов в атрибутах файла называются *режимом доступа к файлу* и представляют права на чтение, запись и выполнение для владельца файла, группы — владельца файла и всех остальных.

Владелец	Группа	Мир
rwx	rwx	rwx

Установленные атрибуты режима `r`, `w` и `x` оказывают определенное влияние на файлы и каталоги, как показано в табл. 9.2.

Таблица 9.2. Атрибуты прав доступа

Атрибут	Файлы	Каталоги
`r`	Разрешается открывать и читать содержимое файла	Разрешается читать содержимое каталога, если вместе с этим атрибутом установлен атрибут права на выполнение
`w`	Разрешается записывать в файл или усекать его; однако этот атрибут не дает права переименовывать и удалять файлы. Возможность переименования и удаления файлов определяется атрибутами вмещающего каталога	Разрешается создавать, удалять и переименовывать файлы внутри каталога, если вместе с этим атрибутом установлен атрибут права на выполнение
`x`	Разрешается интерпретировать файл как программу и выполнять ее. Файлы, содержащие программы на языках сценариев, дополнительно должны быть доступны для чтения, иначе они не будут выполняться	Разрешается входить в каталог, то есть выполнять команду `cd` для перехода в него

В табл. 9.3 приводится несколько примеров установки атрибутов файлов.

Таблица 9.3. Примеры установки атрибутов прав доступа к файлам

Атрибуты файлов	Значение
`-rwx------`	Обычный файл, доступный владельцу для чтения, записи и выполнения. Никто другой не имеет прав доступа к файлу
`-rw-------`	Обычный файл, доступный владельцу для чтения и записи. Никто другой не имеет прав доступа к файлу
`-rw-r--r--`	Обычный файл, доступный владельцу для чтения и записи. Члены группы имеют право читать файл. Все остальные имеют право читать файл
`-rwxr-xr-x`	Обычный файл, доступный владельцу для чтения, записи и выполнения. Все остальные имеют право читать и выполнять файл
`-rw-rw----`	Обычный файл, доступный для чтения и записи только владельцу и членам группы
`Lrwxrwxrwx`	Символическая ссылка. Все символические ссылки имеют недействительные значения атрибутов. Фактические права доступа к файлу определяются атрибутами самого файла, на который указывает символическая ссылка

Таблица 9.3 (*окончание*)

Атрибуты файлов	Значение
`drwxrwx---`	Каталог. Владелец и члены группы могут входить в каталог, создавать, переименовывать и удалять файлы внутри каталога
`drwxr-x---`	Каталог. Владелец может входить в каталог, создавать, переименовывать и удалять файлы внутри каталога. Члены группы могут входить в каталог, но не могут создавать, переименовывать и удалять файлы внутри каталога

chmod — изменение режима доступа к файлу

Для изменения режима (прав) доступа к файлу или каталогу используется команда `chmod`. Имейте в виду, что права доступа к файлу или каталогу может изменить только владелец или суперпользователь. Команда `chmod` поддерживает два разных способа изменения режима:

- с использованием восьмеричных чисел;
- с использованием символического представления.

Сначала рассмотрим использование восьмеричных чисел. При использовании восьмеричной формы записи шаблон желаемых привилегий определяется восьмеричными числами. Так как каждая цифра в восьмеричном числе определяется тремя двоичными разрядами, она точно укладывается в схему хранения режима доступа к файлу. В табл. 9.4 поясняется, что мы имеем в виду.

Таблица 9.4. Режимы доступа к файлу в двоичном и восьмеричном представлениях

Восьмеричное	Двоичное	Режим доступа
0	000	`---`
1	001	`--x`
2	010	`-w-`
3	011	`-wx`
4	100	`r--`
5	101	`r-x`
6	110	`rw-`
7	111	`rwx`

А ПОЧЕМУ ИМЕННО ВОСЬМЕРИЧНОЕ?

Восьмеричная (по основанию 8) и родственная ей *шестнадцатеричная* (по основанию 16) системы счисления часто используются для представления чисел в компьютерах. Мы, люди, рождаемся с десятью пальцами на руках (по крайней мере большинство из нас), поэтому для счета используем систему счисления с основанием 10. Компьютеры, напротив, рождаются с одним пальцем и потому используют для вычисления *двоичную* систему счисления (по основанию 2). Их числа состоят всего из двух цифр, нуля и единицы. Поэтому в двоичной системе счет выглядит так:

```
0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011...
```

В восьмеричной системе используются цифры от нуля до семи:

```
0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21...
```

В шестнадцатеричной системе используются цифры от нуля до девяти плюс буквы от *A* до *F*:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13...
```

В двоичной системе счисления еще можно увидеть смысл (поскольку компьютеры имеют лишь один палец), но в чем польза восьмеричной и шестнадцатеричной систем счисления? Они были придуманы для удобства человека. Очень часто небольшие порции данных представляются в компьютерах битовыми шаблонами. Примером может служить представление цвета в формате RGB. В большинстве дисплеев компьютеров цвет каждого пиксела определяется тремя цветовыми составляющими: 8 бит для красного цвета, 8 бит для зеленого и 8 бит для синего. Красивый сине-голубой цвет можно представить в виде 24-разрядного числа:

```
010000110110111111001101.
```

Хотели бы вы видеть и читать такие числа весь день? Я так не думаю. Именно в таких случаях на выручку приходят другие системы счисления. Каждая цифра в шестнадцатеричной системе счисления представляет четыре двоичные цифры. В восьмеричной системе каждой цифре соответствуют три двоичные цифры. То есть 24-значное значение сине-голубого цвета можно сжать до 6-значного шестнадцатеричного числа: 436FCD. Поскольку цифры в шестнадцатеричных числах «выстраиваются в ряд» с битами в двоичных числах, можно заметить, что красный компонент нашего цвета имеет значение 43, зеленый — 6F и синий — CD.

В наше время шестнадцатеричная форма записи получила большее распространение, чем восьмеричная, но, как будет показано ниже, восьмеричная форма записи все еще оказывается весьма полезной для представления групп из трех двоичных битов.

С помощью трех восьмеричных цифр мы можем определить режим доступа к файлу для владельца, для группы и для остального мира.

```
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me    me   0 2018-03-06 14:52 foo.txt
[me@linuxbox ~]$ chmod 600 foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw------- 1 me    me   0 2018-03-06 14:52 foo.txt
```

Передав аргумент 600, мы установили права для владельца, позволяющие ему читать данные из файла и записывать их в файл, и при этом отобрали все права у группы и остального мира. Несмотря на кажущееся неудобство необходимости запоминания соответствий между восьмеричными и двоичными представлениями, вам, скорее всего, придется использовать лишь несколько наиболее популярных шаблонов: 7 (`rwx`), 6 (`rw-`), 5 (`r-x`), 4 (`r--`) и 0 (`---`).

Команда `chmod` поддерживает также символическую форму определения режимов доступа к файлу. Символическая форма записи делится на три части:

- для кого устанавливаются разрешения;
- какие операции с разрешениями будут выполняться;
- на какие разрешения эти операции будут влиять.

Чтобы указать, для кого устанавливаются разрешения, используется комбинация символов `u`, `g`, `o` и `a`, как показано в табл. 9.5.

Таблица 9.5. Символическая форма записи аргументов команды chmod

Символ	Значение
u	Сокращенно от *user* (пользователь), означает владельца файла или каталога
g	Группа
o	Сокращенно от *others* (другие, остальные), означает весь остальной мир
a	Сокращенно от *all* (все); комбинация из всех трех символов: `u`, `g` и `o`

Если не указан ни один символ, предполагается `a` (all — все). Операцией может быть знак `+`, соответствующий добавлению заданных разрешений, знак `-`, соответствующий отъему заданных разрешений, или знак `=`, указывающий, что только заданные разрешения должны быть установлены, а все остальные отобраны.

Разрешения определяются символами r, w и x. В табл. 9.6 перечислены некоторые примеры символической формы записи.

Таблица 9.6. Примеры символической формы записи прав доступа к файлам

Атрибуты файлов	Значение
u+x	Добавляет право на выполнение, но только для владельца
u-x	Отнимает право на выполнение у владельца
+x	Добавляет право на выполнение для владельца, группы и остального мира. Эквивалент записи a+x
o-rw	Отнимает право на чтение и запись у всех, кроме владельца и группы
go=rw	Устанавливает право на чтение и запись для всех, кроме владельца. Если прежде файл имел разрешение на выполнение для группы и всего мира, это право отнимается
u+x,go=rx	Добавляет право на выполнение для владельца и устанавливает право на чтение и выполнение для группы и всего мира. При выполнении сразу нескольких операций с привилегиями они должны разделяться запятой

Кто-то предпочитает пользоваться восьмеричной формой записи, кому-то больше нравится символическая. Символическая форма записи удобна тем, что позволяет установить единственный атрибут, не влияя на остальные.

Дополнительную информацию и полный список параметров команды chmod можно найти на странице справочного руководства (man). А теперь несколько слов о параметре --recursive: он воздействует и на файлы, и на каталоги, поэтому он не так полезен, как можно было бы предположить, потому что редко требуется устанавливать одинаковые разрешения для файлов и каталогов.

Установка режима доступа к файлу с помощью графического интерфейса

Теперь, ознакомившись с тем, как устанавливаются разрешения для файлов и каталогов, вы лучше поймете диалоги установки разрешений в графическом интерфейсе. В Files (GNOME) и Dolphin (KDE) можно щелкнуть правой кнопкой мыши на файле или на каталоге и вывести диалог со свойствами. На рис. 9.1 изображен такой диалог из GNOME. Здесь вы видите, какие разрешения установлены для владельца, группы и остального мира.

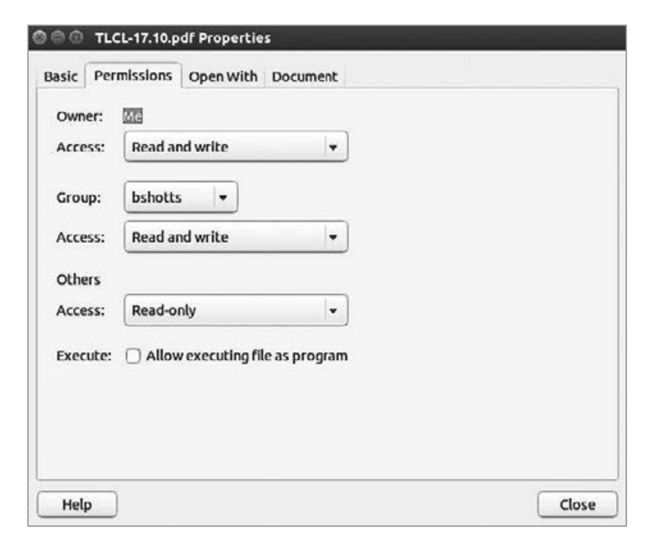


Рис. 9.1. Диалог с правами доступа к файлу в GNOME

umask — определение разрешений доступа к файлам по умолчанию

Команда `umask` определяет разрешения по умолчанию, которые устанавливаются для файла при его создании. В ней с помощью восьмеричной формы записи определяется битовая *маска* для сбрасываемых атрибутов режима доступа. Взгляните:

```
[me@linuxbox ~]$ rm -f foo.txt
[me@linuxbox ~]$ umask
0002
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-r-- 1 me    me   0 2018-03-06 14:53 foo.txt
```

Сначала мы удалили существующий файл *foo.txt*, чтобы, так сказать, начать с чистого листа. Далее мы выполнили команду `umask` без аргумента, чтобы увидеть текущее значение маски. Она вернула нам значение `0002` (часто также используется значение `0022`) — восьмеричное представление действующей маски. Затем мы создали новый файл *foo.txt* и вывели для него разрешения.

Как видите, владелец и группа получили права на чтение и запись, тогда как все остальные — только право на чтение. Весь мир не получил права на запись из-за значения маски. Давайте повторим пример, но на этот раз определим свою маску:

```
[me@linuxbox ~]$ rm foo.txt
[me@linuxbox ~]$ umask 0000
[me@linuxbox ~]$ > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-rw-rw- 1 me    me   0 2018-03-06 14:58 foo.txt
```

После установки маски в значение `0000` (таким способом мы фактически выключили ее) вновь созданный файл получил разрешение на запись для всего мира. Чтобы лучше понять суть происходящего, мы снова должны вернуться к восьмеричным числам. Если развернуть маску в двоичное представление и сравнить ее с двоичным представлением атрибутов, можно понять, что произошло:

Исходный режим доступа к файлу	`--- rw- rw- rw-`
Маска	`000 000 000 010`
Результат	`--- rw- rw- r--`

Забудем пока про начальные нули (мы вернемся к ним чуть позже) и обратим внимание, что атрибут, соответствующий той позиции, где в маске стоит 1, был сброшен, — в данном случае право на запись для всего мира. Теперь понятно, что делает маска. В любой позиции, где в маске появляется 1, соответствующий атрибут сбрасывается. Если посмотреть на значение маски `0022`, легко увидеть, что оно делает:

Исходный режим доступа к файлу	`--- rw- rw- rw-`
Маска	`000 000 010 010`
Результат	`--- rw- r-- r--`

И снова атрибуты, соответствующие позициям, где в маске стоит 1, были сброшены. Поэкспериментируйте с другими значениями (попробуйте несколько 7), чтобы лучше усвоить, как действует маска. Закончив эксперименты, не забудьте все вернуть в исходное состояние:

```
[me@linuxbox ~]$ rm foo.txt; umask 0002
```

Вам редко придется изменять маску, потому что значение по умолчанию, устанавливаемое дистрибутивом, прекрасно подходит для большинства нужд. Однако в некоторых ситуациях, требующих повышенной безопасности, маску может понадобиться изменить.

Некоторые специальные разрешения

Обычно разрешения в восьмеричном представлении мы видим как трехзначные числа, но технически более правильно выражать их четырехзначными числами. Почему? Потому что в дополнение к разрешениям на чтение, запись и выполнение существует еще несколько редко используемых разрешений.

Первый атрибут — *бит setuid* (восьмеричное значение 4000). Если это разрешение применяется к выполняемому файлу, в качестве *эффективного идентификатора пользователя* для процесса устанавливается не идентификатор реального пользователя (пользователя, фактически запустившего программу), а идентификатор владельца программы. Чаще этот бит устанавливается для программ, владельцем которых является суперпользователь. Когда обычный пользователь запускает программу с установленным битом *setuid*, принадлежащую пользователю *root*, программа выполняется с эффективными привилегиями суперпользователя. Это дает возможность программам обращаться к файлам и каталогам, недоступным для обычного пользователя. Очевидно, что из-за возникающих проблем безопасности число таких программ в системе должно быть сведено к минимуму.

Второй редко используемый атрибут — *бит setgid* (восьмеричное значение 2000). По аналогии с битом *setuid* он устанавливает *эффективный идентификатор группы* для процесса, выбирая вместо идентификатора группы реального пользователя группу владельца файла. Если установить бит *setgid* для каталога, вновь создаваемые файлы в этом каталоге будут принадлежать группе владельца каталога, а не группе владельца файла, создавшего его. Это разрешение может пригодиться для установки на каталоги, содержимое которых должно быть доступно всем членам основной группы владельца каталога, независимо от принадлежности к основной группе владельца файла.

Третий атрибут называется *битом sticky* (восьмеричное значение 1000). Это пережиток, оставшийся от первых версий Unix, которые предоставляли возможность пометить выполняемый файл как «невытесняемый». Linux игнорирует бит *sticky* у файлов, но если установить его для каталога, он не позволит пользователю удалять или переименовывать файлы, если только пользователь не является владельцем каталога, владельцем файла или суперпользователем. Это разрешение часто применяется для управления доступом к общим каталогам, таким как */tmp*.

Ниже приводится несколько примеров использования `chmod` с символической формой определения этих специальных разрешений. Первый пример — установка бита *setuid* на файл программы:

```
chmod u+s program
```

Далее — установка бита *setgid* на каталог:

```
chmod g+s dir
```

Наконец, установка бита *sticky* на каталог:

```
chmod +t dir
```

Специальные разрешения мы видим в выводе команды `ls`. Ниже приводится несколько примеров. Первый — программа с битом *setuid*:

```
-rwsr-xr-x
```

Теперь — каталог с атрибутом *setgid*:

```
drwxrwsr-x
```

Наконец, каталог с битом *sticky*:

```
drwxrwxrwt
```

Изменение идентичности

Время от времени возникает необходимость приобрести идентичность другого пользователя. Обычно это требуется для получения привилегий суперпользователя, чтобы выполнить некоторые административные задачи, но точно так же можно «превратиться» в другого обычного пользователя, чтобы, к примеру, проверить настройки учетной записи. Существует три способа приобрести альтернативную идентичность:

1. Выйти из системы и войти вновь с учетными данными другого пользователя.
2. Воспользоваться командой `su`.
3. Воспользоваться командой `sudo`.

Мы пропустим первый способ, потому что уже знаем, как им воспользоваться, и он не так удобен, как два других. В рамках сеанса работы с командной оболочкой команда `su` позволяет приобрести идентичность другого пользователя и либо начать новый сеанс командной оболочки с идентификатором этого пользователя, либо запустить одиночную команду от его имени. Команда `sudo` позволяет администратору записать настройки в конфигурационный файл с именем */etc/sudoers* и определить конкретные команды, которые сможет выполнять тот или иной пользователь под приобретенной идентичностью. Выбор между `su` и `sudo` в значительной степени определяется используемым дистрибутивом Linux. Большинство

дистрибутивов включают обе команды, но в настройках предпочтение отдается той или иной. Начнем с команды su.

su — запуск командной оболочки с подстановкой идентификаторов пользователя и группы

Команда su используется для запуска нового сеанса работы с командной оболочкой от имени другого пользователя. Команда имеет следующий синтаксис:

```
su [-[l]] [пользователь]
```

Если указан параметр -l, запущенная командная оболочка станет оболочкой входа для указанного пользователя. Это означает, что будет загружено окружение пользователя и текущим рабочим каталогом станет домашний каталог пользователя. Часто это именно то, что требуется. Если пользователь не указан, подразумевается суперпользователь. Обратите внимание, что (довольно необычно) параметр -l можно сократить до -, и эта особенность часто используется на практике. Запустить командную оболочку от имени суперпользователя можно следующим образом:

```
[me@linuxbox ~]$ su -
Пароль:
[root@linuxbox ~]#
```

После ввода команды будет запрошен пароль суперпользователя. После ввода правильного пароля появится новое приглашение к вводу, показывающее, что данная командная оболочка обладает привилегиями суперпользователя (символ # в конце вместо символа $) и текущим рабочим каталогом теперь стал домашний каталог суперпользователя (обычно */root*). После запуска новой оболочки можно выполнять команды с привилегиями суперпользователя. Завершим работу, введя команду exit, чтобы вернуться в предыдущую командную оболочку:

```
[root@linuxbox ~]# exit
[me@linuxbox ~]$
```

С помощью su можно так же просто выполнить единственную команду, не запуская новый интерактивный сеанс:

```
su -c 'команда'
```

При использовании этой формы команде su передается единственная командная строка для выполнения. Не забудьте заключить команду в кавычки, чтобы

предотвратить дополнительную ее интерпретацию механизмами подстановки текущей командной оболочки:

```
[me@linuxbox ~]$ su -c 'ls -l /root/*'
Пароль:
-rw------- 1 root root     754 2007-08-11 03:19 /root/anaconda-ks.cfg

/root/Mail:
итого 0
[me@linuxbox ~]$
```

sudo — выполнение команды от имени другого пользователя

Команда `sudo` во многом подобна команде `su`, но имеет некоторые важные дополнительные особенности. Администратор может определить порядок использования `sudo` обычными пользователями, ограничив возможность запуска команд от имени другого пользователя (обычно суперпользователя). В частности, пользователю может быть разрешен доступ к одним командам и запрещен к другим.

Еще одно важное отличие состоит в том, что `sudo` не требует ввода пароля суперпользователя. Для аутентификации в команде `sudo` пользователь должен ввести свой пароль. Например, допустим, что настройки `sudo` позволяют выполнить некоторую мифическую программу резервного копирования с именем `backup_script`, требующую привилегий суперпользователя. С помощью `sudo` ее можно запустить так:

```
[me@linuxbox ~]$ sudo backup_script
Пароль:
System Backup Starting...
```

После ввода команды вам будет предложено ввести пароль (ваш, а не суперпользователя), и по завершении аутентификации указанная команда будет выполнена. Одно важное отличие между `su` и `sudo` — последняя не запускает новую командную оболочку и не загружает окружение другого пользователя. Это означает, что команды не требуется экранировать как-то иначе, чем при запуске той же команды без использования `sudo`. Имейте в виду, что такое ее поведение можно переопределить с помощью различных параметров. Также отметьте, что `sudo` можно использовать для запуска интерактивного сеанса с привилегиями суперпользователя (почти так же, как командой `su -`), передав ей параметр `-i`. Подробности ищите на странице справочного руководства (man) для sudo.

Чтобы увидеть, какие привилегии дает команда `sudo`, вызовите ее с параметром `-l`:

```
[me@linuxbox ~]$ sudo -l
User me may run the following commands on this host:
    (ALL) ALL
```

UBUNTU И SUDO

Обычные пользователи иногда сталкиваются с необходимостью выполнить некоторую операцию, требующую привилегий суперпользователя. К числу таких операций относится установка и обновление программного обеспечения, правка системных конфигурационных файлов и доступ к устройствам. В мире Windows эта проблема часто решается передачей пользователям административных привилегий, что позволяет им решать подобные задачи. Однако программы, запускаемые такими пользователями, получают те же привилегии. В большинстве случаев это именно то, что нужно, но это также дает возможность беспрепятственной работы вредоносному программному обеспечению, такому как вирусы.

В мире Unix, вследствие многопользовательской природы этой операционной системы, всегда проводилась четкая грань между обычными пользователями и администраторами. Идеология Unix заключается в том, чтобы предоставлять привилегии суперпользователя, только когда они действительно необходимы. Для этого часто используются команды `su` и `sudo`.

Еще несколько лет тому назад большинство дистрибутивов Linux использовали с этой целью команду `su`. Команда `su` не требует настройки, как команда `sudo`, а наличие учетной записи root — давняя традиция в Unix. Вместе это порождает проблему. Пользователи могут испытывать соблазн действовать от имени root без всякой необходимости. Фактически некоторые пользователи вообще работают в своих системах, регистрируясь исключительно как root, чтобы избежать появления раздражающих сообщений «permission denied» (доступ запрещен). Такой подход ухудшает защищенность Linux, низводя ее до уровня Windows. Не самое лучшее решение.

Создатели Ubuntu предприняли иной подход. По умолчанию Ubuntu запрещает регистрироваться в системе с учетной записью root (не позволяя устанавливать пароль для этой учетной записи), а для получения привилегий суперпользователя предлагает использовать `sudo`. Начальная учетная запись пользователя обладает полным доступом к привилегиям суперпользователя через `sudo` и может наделять аналогичными привилегиями другие, вновь создаваемые учетные записи.

chown — изменение владельца и группы файла

Команда chown используется для изменения владельца и группы файла или каталога. Для использования этой команды необходимы привилегии суперпользователя. Команда chown имеет следующий синтаксис:

```
chown [владелец][:[группа]] файл...
```

chown может изменить владельца и/или группу файла в зависимости от первого аргумента. В табл. 9.7 приводится несколько примеров команды.

Таблица 9.7. Примеры аргументов команды chown

Аргумент	Результаты
bob	Изменит принадлежность файла, назначив владельцем пользователя bob
bob:users	Изменит принадлежность файла, назначив владельцем пользователя bob и группу users
:admins	Изменит принадлежность файла, назначив группу admins
bob:	Изменит принадлежность файла, назначив владельцем пользователя bob и группу этого пользователя

Представьте, что существуют два пользователя: janet, имеющий доступ к привилегиям суперпользователя, и tony, лишенный таких привилегий. Пользователю janet нужно скопировать файл из своего домашнего каталога в домашний каталог пользователя tony. Поскольку пользователь janet хочет, чтобы пользователь tony смог редактировать файл, janet должен изменить владельца скопированного файла, назначив владельцем tony:

```
[janet@linuxbox ~]$ sudo cp myfile.txt ~tony
Пароль:
[janet@linuxbox ~]$ sudo ls -l ~tony/myfile.txt
-rw-r--r-- 1  root  root  8031 2018-03-20 14:30 /home/tony/myfile.txt
[janet@linuxbox ~]$ sudo chown tony: ~tony/myfile.txt
[janet@linuxbox ~]$ sudo ls -l ~tony/myfile.txt
-rw-r--r-- 1  tony  tony  8031 2018-03-20 14:30 /home/tony/myfile.txt
```

Здесь видно, как пользователь janet копирует файл из своего каталога в домашний каталог пользователя tony. Далее janet заменяет владельца файла root (результат использования sudo) на tony. Добавив двоеточие в конец первого аргумента, janet

одновременно изменяет группу, которой принадлежит файл, на основную группу пользователя `tony`, которая, так уж получилось, носит то же имя `tony`.

Заметили ли вы, что после первого использования команда `sudo` не предложила пользователю `janet` вновь ввести пароль? Это объясняется тем, что в большинстве конфигураций `sudo` продолжает «доверять» пользователю в течение нескольких минут (пока не истечет время ее действия).

chgrp — изменение группы файла

В старых версиях Unix команда `chown` изменяла только владельца файла, но не группу. Чтобы изменить группу, предоставлялась другая команда, `chgrp`. Она действует практически так же, как `chown`, но имеет больше ограничений.

Использование привилегий

Теперь, когда мы разобрались, как действует механизм привилегий, самое время научиться пользоваться ими. Далее демонстрируется решение типичной задачи — настройка общего каталога. Представьте себе двух пользователей, `bill` и `karen`. Оба имеют коллекции музыкальных произведений и хотели бы настроить общий каталог, где могли бы хранить файлы в формате Ogg Vorbis или MP3. Пользователь `bill` имеет доступ к привилегиям суперпользователя через `sudo`.

Первое, что нужно сделать, — это создать группу, куда будут входить оба пользователя, `bill` и `karen`. С помощью графического инструмента для управления пользователями `bill` создает группу с именем `music` и добавляет в нее пользователей `bill` и `karen`, как показано на рис. 9.2.

Рис. 9.2. Создание новой группы

Далее `bill` создает каталог для музыкальных файлов:

```
[bill@linuxbox ~]$ sudo mkdir /usr/local/share/Music
Пароль:
```

Поскольку `bill` манипулирует файлами за пределами своего домашнего каталога, ему необходимы привилегии суперпользователя. После создания каталог получает следующие права доступа и владельца:

```
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxr-xr-x 2 root root 4096 2018-03-21 18:05 /usr/local/share/Music
```

Как видите, каталогом владеет `root`, и для него установлен режим доступа 755. Чтобы сделать каталог общим, `bill` должен изменить группу каталога и права доступа для группы:

```
[bill@linuxbox ~]$ sudo chown :music /usr/local/share/Music
[bill@linuxbox ~]$ sudo chmod 775 /usr/local/share/Music
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxrwxr-x 2 root music 4096 2018-03-21 18:05 /usr/local/share/Music
```

И что все это означает? А означает это следующее: владельцем каталога */usr/local/share/Music* является `root`, и члены группы `music` получают права на запись и чтение в этом каталоге. Группа `music` включает пользователей `bill` и `karen`; то есть `bill` и `karen` могут создавать файлы в каталоге */usr/local/share/Music*. Другие пользователи могут просматривать содержимое каталога, но не могут создавать файлы в нем.

Но остается нерешенной еще одна проблема. С текущими разрешениями файлы и каталоги внутри каталога *Music* будут создаваться с обычными разрешениями для пользователей `bill` и `karen`:

```
[bill@linuxbox ~]$ > /usr/local/share/Music/test_file
[bill@linuxbox ~]$ ls -l /usr/local/share/Music
-rw-r--r-- 1 bill   bill   0 2018-03-24 20:03 test_file
```

В действительности здесь наблюдаются две проблемы. Во-первых, маска `umask` в этой системе имеет значение 0022, что не позволяет членам группы записывать в файлы, принадлежащие другим членам группы. Это не проблема, если общий каталог хранит только файлы, но так как в данном каталоге предполагается хранить музыкальные произведения, а музыкальные произведения обычно принято организовывать в иерархии по исполнителям и альбомам, членам группы может понадобиться создавать файлы в каталогах, принадлежащих другим членам. Нам нужно изменить маску `umask` для пользователей `bill` и `karen` на 0002.

Во-вторых, каждый файл и каталог, созданный одним членом группы, будет принадлежать основной группе пользователя, а не группе music. Исправить этот недостаток можно установкой бита *setgid* на каталог:

```
[bill@linuxbox ~]$ sudo chmod g+s /usr/local/share/Music
[bill@linuxbox ~]$ ls -ld /usr/local/share/Music
drwxrwsr-x 2 root music 4096 2018-03-24 20:03 /usr/local/share/Music
```

Теперь можно проверить, устранили ли проблему вновь добавленные разрешения. bill устанавливает маску umask в значение 0002, удаляет предыдущий проверочный файл и создает новый проверочный файл и каталог:

```
[bill@linuxbox ~]$ umask 0002
[bill@linuxbox ~]$ rm /usr/local/share/Music/test_file
[bill@linuxbox ~]$ > /usr/local/share/Music/test_file
[bill@linuxbox ~]$ mkdir /usr/local/share/Music/test_dir
[bill@linuxbox ~]$ ls -l /usr/local/share/Music
drwxrwsr-x 2 bill   music 4096 2018-03-24 20:24 test_dir
-rw-rw-r-- 1 bill   music    0 2018-03-24 20:22 test_file
[bill@linuxbox ~]$
```

И файл и каталог теперь созданы с правильными правами доступа, позволяющими всем членам группы music создавать файлы и каталоги внутри каталога *Music*.

Осталась только проблема с маской umask. Дело в том, что установленная маска действует лишь до конца сеанса и сбрасывается по его завершении. В главе 11 мы узнаем, как сохранить действие измененной маски umask между сеансами.

Изменение своего пароля

Последняя тема этой главы: изменение собственного пароля (и паролей других пользователей при наличии привилегий суперпользователя). Для установки и изменения пароля используется команда passwd. Она имеет следующий синтаксис:

```
passwd [пользователь]
```

Чтобы изменить свой пароль, просто введите команду passwd. Вам будет предложено ввести старый, а затем новый пароль:

```
[me@linuxbox ~]$ passwd
Смена пароля для me.
(текущий) пароль UNIX:
Введите новый пароль UNIX:
```

Команда пытается вынудить пользователей вводить «сильные» пароли. Это означает, что она будет отвергать слишком короткие пароли, очень похожие на предыдущие пароли, пароли, являющиеся словарными словами или легко угадываемые:

```
[me@linuxbox ~]$ passwd
Смена пароля для me.
(текущий) пароль UNIX:
Введите новый пароль UNIX:
BAD PASSWORD: is too similar to the old one
Введите новый пароль UNIX:
Выберите пароль большей длины
Введите новый пароль UNIX:
BAD PASSWORD: it is based on a dictionary word
```

При наличии привилегий суперпользователя можно передать команде `passwd` аргумент с именем пользователя, чтобы установить пароль для этого пользователя. Суперпользователю доступна также возможность блокировки учетных записей, установки времени действия пароля и многое другое. За подробностями обращайтесь к странице справочного руководства (man) для команды `passwd`.

Заключение

В этой главе мы увидели, как в Unix-подобных системах, таких как Linux, осуществляется управление разрешениями, дающими пользователям права доступа на чтение, запись и выполнение к файлам и каталогам. Основные идеи этой системы разрешений были выработаны на самых ранних этапах развития Unix и выдержали испытание временем. Однако следует заметить, что традиционный механизм разрешений в Unix-подобных системах не обладает высокой избирательностью аналогичных механизмов в более современных системах.

10 Процессы

Современные операционные системы обычно являются *многозадачными*, в том смысле, что создают иллюзию одновременного решения множества задач, быстро переключаясь с выполнения одной программы на другую. Ядро Linux управляет всем этим посредством *процессов*. Именно с помощью процессов Linux организует приостановку программ в ожидании, пока наступит их очередь использовать процессор.

Иногда компьютер становится «вялым», или приложение вообще перестает откликаться на команды пользователя. Сейчас мы познакомимся с некоторыми инструментами командной строки, позволяющими увидеть, что делают программы, и завершить процессы, вышедшие из-под контроля.

В этой главе будут представлены следующие команды:

`ps` — выводит список процессов, выполняющихся в текущий момент;

`top` — выводит задачи;

`jobs` — выводит список активных заданий;

`bg` — переводит задание в фоновый режим работы;

`fg` — переводит задание в режим работы на переднем плане;

`kill` — посылает сигнал процессу;

`killall` — останавливает процессы по именам;

`shutdown` — останавливает или перезагружает систему.

Как действует процесс

В момент запуска системы ядро инициирует выполнение нескольких собственных задач в виде процессов и запускает программу с названием `init`. В свою очередь

`init` выполняет последовательность сценариев командной оболочки (находятся в */etc*), называемых *сценариями начальной загрузки* (init scripts), которые запускают все системные службы. Многие из этих служб реализованы как *программы-демоны* (daemon programs), то есть программы, действующие в фоновом режиме и выполняющие свою работу без участия пользователя. Поэтому, даже в отсутствие зарегистрированных пользователей, система выполняет определенные служебные процедуры.

Принцип, по которому программа может запускать другие программы, выражается правилом: *родительский процесс* запускает *дочерний процесс*.

Ядро хранит информацию обо всех процессах, чтобы упорядочить их работу. Например, каждому процессу присваивается номер, который называют *идентификатором процесса* (Process ID, PID). Идентификаторы процессов присваиваются в порядке возрастания, при этом процесс `init` всегда получает идентификатор PID 1. Ядро также следит за памятью, выделенной каждому процессу, и за готовностью процессов возобновить выполнение. Подобно файлам, процессы также имеют идентификаторы владельца и пользователя, эффективный (или действующий) идентификатор пользователя и т. д.

Просмотр списка процессов

Чаще всего для просмотра списка процессов используется команда `ps` (существуют и другие команды). Программа `ps` имеет множество параметров, но в самом простейшем случае она используется следующим образом:

```
[me@linuxbox ~]$ ps
  PID TTY          TIME CMD
 5198 pts/1    00:00:00 bash
10129 pts/1    00:00:00 ps
```

В этом примере команда вывела список с двумя процессами: процесс 5198 и процесс 10129 — программы `bash` и `ps` соответственно. Как можно заметить, по умолчанию `ps` выводит не очень много информации, только процессы, связанные с текущим сеансом. Чтобы увидеть больше, следует передать дополнительные параметры, но прежде чем мы сделаем это, давайте рассмотрим другие поля в выводе команды `ps`. Поле TTY — это сокращение от *teletype* (телетайп), оно содержит информацию об *управляющем терминале* процесса. В Unix в этом поле выводится тип терминала. Поле TIME содержит объем процессорного времени, потребленного процессом. Как видите, ни один из процессов не является слишком обременительным для компьютера.

Если добавить параметр `x`, можно получить более богатую информацию о происходящем в системе:

```
[me@linuxbox ~]$ ps x
  PID TTY      STAT   TIME COMMAND
 2799 ?        Ssl    0:00 /usr/libexec/bonobo-activation-server –ac
 2820 ?        Sl     0:01 /usr/libexec/evolution-data-server-1.10 --
15647 ?        Ss     0:00 /bin/sh /usr/bin/startkde
15751 ?        Ss     0:00 /usr/bin/ssh-agent /usr/bin/dbus-launch --
15754 ?        S      0:00 /usr/bin/dbus-launch --exit-with-session
15755 ?        Ss     0:01 /bin/dbus-daemon --fork --print-pid 4 –pr
15774 ?        Ss     0:02 /usr/bin/gpg-agent -s –daemon
15793 ?        S      0:00 start_kdeinit --new-startup +kcminit_start
15794 ?        Ss     0:00 kdeinit Running...
15797 ?        S      0:00 dcopserver –nosid
и еще много других процессов...
```

Дополнительный параметр x (обратите внимание на отсутствие дефиса) сообщает команде ps, что та должна вывести все процессы, независимо от того, какие терминалы (если таковые имеются) управляют ими. Символ ? в поле TTY указывает на отсутствие управляющего терминала. Таким образом, параметр x позволяет увидеть все процессы в системе, которыми мы владеем.

Так как в системе одновременно выполняется множество процессов, ps производит довольно длинные списки. Часто бывает полезно передать вывод ps команде less через конвейер, чтобы его проще было просматривать. Некоторые комбинации параметров приводят к выводу очень длинных строк, поэтому нелишним будет также распахнуть окно эмулятора терминала на весь экран.

В этом примере в выводе появился новый столбец — STAT. Название STAT — это сокращение от *state* (состояние), столбец содержит информацию о текущем состоянии процесса, как показано в табл. 10.1.

Таблица 10.1. Состояния процессов

Состояние	Значение
R	Выполняется. Процесс выполняется или готов к выполнению
S	Приостановлен. Процесс временно не выполняется; скорее всего, находится в ожидании определенного события, такого как нажатие клавиши или прибытие сетевого пакета
D	Приостановлен без возможности прерывания. Процесс ожидает завершения операции ввода/вывода, например, дисковым устройством
T	Остановлен. Процесс принудительно остановлен (подробнее об этом рассказывается ниже)
Z	Недействующий процесс-«зомби». Это дочерний процесс, который завершился, но не был удален родителем

Состояние	Значение
<	Высокоприоритетный процесс. Существует возможность наиболее важным процессам выделить больше процессорного времени. Данное свойство процесса называется niceness (уступчивость). Про процессы с более высокими приоритетами говорят, что они менее уступчивы, потому что потребляют больше процессорного времени, оставляя меньше другим процессам
N	Низкоприоритетный процесс. Процесс с низким приоритетом (или уступчивый процесс) получает процессорное время только после того, как будут обслужены процессы с более высоким приоритетом

Символ, описывающий состояние процесса, может сопровождаться другими символами. Они отражают некоторые экзотические характеристики процессов. За дополнительной информацией обращайтесь к странице справочного руководства (man) для `ps`.

Еще одна популярная комбинация параметров — `aux` (без дефиса в начале). Она позволяет получить еще больше информации:

```
[me@linuxbox ~]$ ps aux
USER     PID %CPU %MEM    VSZ   RSS TTY      STAT START    TIME COMMAND
root       1 0.0  0.0    2136   644 ?        Ss Mar05      0:31 init
root       2 0.0  0.0       0     0 ?        S< Mar05      0:00 [kt]
root       3 0.0  0.0       0     0 ?        S< Mar05      0:00 [mi]
root       4 0.0  0.0       0     0 ?        S< Mar05      0:00 [ks]
root       5 0.0  0.0       0     0 ?        S< Mar05      0:06 [wa]
root       6 0.0  0.0       0     0 ?        S< Mar05      0:36 [ev]
root       7 0.0  0.0       0     0 ?        S< Mar05      0:00 [kh]
и еще много других процессов...
```

Эта комбинация параметров выводит процессы, принадлежащие всем пользователям. При использовании параметров без начального дефиса команда действует «в стиле BSD». Linux-версия команды `ps` может имитировать поведение программы `ps`, используемой в некоторых реализациях Unix. С помощью этих параметров мы получили дополнительные столбцы, описанные в табл. 10.2.

Таблица 10.2. Заголовки столбцов при выполнении ps в стиле BSD

Заголовок	Значение
USER	Идентификатор пользователя. Это владелец процесса
%CPU	Использование процессора в процентах
%MEM	Использование памяти в процентах

Таблица 10.2 (*окончание*)

Заголовок	Значение
VSZ	Объем виртуальной памяти
RSS	Размер страниц памяти. Объем физической памяти (ОЗУ), используемой процессом, кб
START	Время запуска процесса. Для значений свыше 24 часов выводится дата

Просмотр состояния процессов в динамике с помощью top

Команда ps сообщает массу информации о том, что делается в компьютере, но она дает только мгновенный снимок, то есть возвращаемая ею информация действительна лишь на момент вызова команды. Чтобы увидеть работу компьютера в динамике, воспользуемся командой top:

```
[me@linuxbox ~]$ top
```

Программа top постоянно обновляет информацию о процессах (по умолчанию с периодом, равным 3 секундам), чтобы показать их активность с течением времени. Имя программы *top* отражает тот факт, что она используется для просмотра «топа» (наиболее активных) процессов в системе. Вывод команды top делится на две части: сводная информация о системе и таблица процессов, отсортированных по потреблению ими процессора:

```
top - 14:59:20 up 6:30,  2 users,  load average: 0.07, 0.02, 0.00
Tasks: 109 total,  1 running, 106 sleeping,   0 stopped,   2 zombie
Cpu(s):  0.7%us, 1.0%sy,  0.0%ni, 98.3%id,  0.0%wa,  0.0%hi,  0.0%si
Mem:    319496k total,  314860k used,     4636k free,    19392k buff
Swap:   875500k total,  149128k used,   726372k free,   114676k cach

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM   TIME+  COMMAND
 6244 me        39  19 31752 3124 2188 S  6.3  1.0 16:24.42 trackerd
11071 me        20   0  2304 1092  840 R  1.3  0.3  0:00.14 top
 6180 me        20   0  2700 1100  772 S  0.7  0.3  0:03.66 dbus-dae
 6321 me        20   0 20944 7248 6560 S  0.7  2.3  2:51.38 multiloa
 4955 root      20   0  104m 9668 5776 S  0.3  3.0  2:19.39 Xorg
    1 root      20   0  2976  528  476 S  0.0  0.2  0:03.14 init
    2 root      15  -5     0    0    0 S  0.0  0.0  0:00.00 kthreadd
    3 root      RT  -5     0    0    0 S  0.0  0.0  0:00.00 migratio
    4 root      15  -5     0    0    0 S  0.0  0.0  0:00.72 ksoftirq
    5 root      RT  -5     0    0    0 S  0.0  0.0  0:00.04 watchdog
    6 root      15  -5     0    0    0 S  0.0  0.0  0:00.42 events/0
```

```
    7 root      15  -5    0    0    0 S  0.0  0.0  0:00.06 khelper
   41 root      15  -5    0    0    0 S  0.0  0.0  0:01.08 kblockd/
   67 root      15  -5    0    0    0 S  0.0  0.0  0:00.00 kseriod
  114 root      20   0    0    0    0 S  0.0  0.0  0:01.62 pdflush
  116 root      15  -5    0    0    0 S  0.0  0.0  0:02.44 kswapd0
```

Раздел со сводной информацией содержит массу интересных сведений. Описание выводимой в этом разделе информации приводится в табл. 10.3.

Таблица 10.3. Поля в разделе со сводной информацией команды top

Строка	Поле	Значение
1	`top`	Имя программы
	`14:59:20`	Текущее время
	`up 6:30`	Это поле называется *uptime* (продолжительность работы). Показывает время, прошедшее с момента последней загрузки системы. В данном примере система проработала 6½ часа
	`2 users`	В системе работают два пользователя
	`load average:`	Средняя нагрузка (load average) — это число процессов, ожидающих возобновления работы; то есть число процессов в состоянии «выполняется» и совместно использующих процессор. Здесь показаны три значения для разных интервалов времени. Первое значение отражает среднюю нагрузку за последние 60 секунд, второе — за последние 5 минут и третье — за последние 15 минут. Значения ниже 1,0 сообщают, что система не нагружена
2	`Tasks:`	Суммарное число процессов в разных состояниях
	`0.7%us`	0,7 % процессорного времени затрачено на выполнение *пользовательских* (user) процессов. Под этим подразумеваются процессы за пределами самого ядра
	`1.0%sy`	1,0 % процессорного времени затрачено на выполнение *системных* (system) процессов (ядра)
	`0.0%ni`	0,0 % процессорного времени затрачено на выполнение уступчивых (nice), то есть низкоприоритетных процессов
	`98.3%id`	98,3 % процессорного времени составили простои
4	`Mem:`	Объем использованной физической памяти (ОЗУ)
5	`Swap:`	Объем использованного пространства в файле подкачки (виртуальная память)

Программа `top` принимает ряд команд с клавиатуры. Наибольший интерес представляет команда `h`, которая выводит экран со справочной информацией, и `q`, которая завершает `top`.

Оба основных окружения рабочего стола включают приложения с графическим интерфейсом, отображающие аналогичную информацию (подобно тому, как это делает Task Manager (Диспетчер задач) в Windows), но я считаю, что `top` лучше своих аналогов с графическим интерфейсом, потому что она работает быстрее и потребляет меньше системных ресурсов. В конце концов, программа мониторинга системы не должна замедлять систему, за которой мы наблюдаем.

Управление процессами

Теперь, когда мы можем видеть процессы и наблюдать за ними, можно приступать к управлению ими. Роль подопытной морской свинки в наших экспериментах исполнит маленькая программка `xlogo`. Программа `xlogo` — это демонстрационная программа, поставляемая в составе X Window System (механизм создания графического изображения на дисплее), которая просто отображает окно с логотипом X. Для начала давайте познакомимся с объектом экспериментов:

```
[me@linuxbox ~]$ xlogo
```

После ввода команды на экране должно появиться небольшое окно с логотипом (рис. 10.1). В некоторых системах `xlogo` может выводить предупреждающее сообщение, но его можно смело игнорировать.



Рис. 10.1. Окно программы xlogo

ПРИМЕЧАНИЕ

Если программа `xlogo` отсутствует в системе, попробуйте вместо нее использовать `gedit` или `kwrite`.

Чтобы убедиться, что `xlogo` работает, попробуйте изменить размеры ее окна. Если после изменения размеров содержимое окна перерисовывается, значит, программа работает.

Заметили ли вы, что командная оболочка не вывела приглашения к вводу после выполнения команды? Это объясняется тем, что командная оболочка ждет, пока программа завершится. То же самое происходило со всеми программами, которые мы запускали до сих пор. Если закрыть окно `xlogo`, оболочка выведет приглашение к вводу.

Прерывание процесса

Давайте понаблюдаем, что происходит после запуска `xlogo`. Сначала введите команду `xlogo` и убедитесь, что программа работает. Затем вернитесь в окно терминала и нажмите комбинацию CTRL+C.

```
[me@linuxbox ~]$ xlogo
[me@linuxbox ~]$
```

Комбинация CTRL+C в терминале *прерывает* выполнение программы. Фактически мы вежливо попросили программу завершиться. После нажатия CTRL+C окно `xlogo` закроется и командная оболочка выведет приглашение к вводу.

Таким способом можно прервать выполнение многих (но не всех) программ командной строки.

Перевод процессов в фоновый режим

Представьте, что нам потребовалось вернуться в командную оболочку, не прерывая выполнения программы `xlogo`. Мы можем сделать это, переведя программу в фоновый режим работы. Считайте, что терминал имеет *передний план* (то, что видно на поверхности, например приглашение к вводу) и *задний план* (фон, то, что скрыто под поверхностью). Чтобы запустить программу сразу в фоновом режиме, нужно добавить в конец команды амперсанд (`&`):

```
[me@linuxbox ~]$ xlogo &
[1] 28236
[me@linuxbox ~]$
```

После ввода такой команды на экране появится окно `xlogo`, а командная оболочка вернется в приглашение к вводу, но перед этим выведет таинственные числа. Это сообщение является частью *механизма управления заданиями* (job control). Таким способом командная оболочка сообщает, что мы запустили задание с номером 1

([1]) и оно получило идентификатор процесса PID 28236. Если теперь выполнить команду ps, можно увидеть этот процесс:

```
[me@linuxbox ~]$ ps
  PID TTY          TIME CMD
10603 pts/1    00:00:00 bash
28236 pts/1    00:00:00 xlogo
28239 pts/1    00:00:00 ps
```

Механизм управления заданиями также дает возможность вывести список заданий, запущенных в терминале. Этот список можно получить командой jobs:

```
[me@linuxbox ~]$ jobs
[1]+ Running                 xlogo &
```

Результаты показывают, что у нас имеется одно выполняющееся задание с номером 1, которое было запущено командой xlogo &.

Возврат процесса на передний план

Процесс в фоновом режиме не получает ввод с клавиатуры, в том числе не видит попыток прервать его комбинацией CTRL+C. Вернуть процесс на передний план можно командой fg, как в следующем примере:

```
[me@linuxbox ~]$ jobs
[1]+ Running                 xlogo &
[me@linuxbox ~]$ fg %1
xlogo
```

За командой fg должен следовать знак процента и номер задания (эта комбинация называется *спецификатором задания*, или jobspec). Если имеется только одно фоновое задание, спецификатор можно опустить. Теперь завершим xlogo вводом CTRL+C.

Приостановка процесса

Иногда необходимо приостановить процесс на время, не завершая его. Это часто делается с целью перевести процесс переднего плана в фоновый режим. Чтобы приостановить процесс переднего плана, используйте комбинацию CTRL+Z. Давайте попробуем. В командной строке введите команду xlogo, нажмите ENTER, а затем комбинацию CTRL+Z:

```
[me@linuxbox ~]$ xlogo
[1]+ Stopped                 xlogo
[me@linuxbox ~]$
```

После приостановки `xlogo` убедимся, что программа действительно приостановилась, для этого попытаемся изменить размер окна `xlogo`. Увы, программа никак не реагирует на наши действия. Далее можно или вернуть программу на передний план командой `fg`, или перевести ее в фоновый режим командой `bg`:

```
[me@linuxbox ~]$ bg %1
[1]+ xlogo &
[me@linuxbox ~]$
```

Так же как в случае с командой `fg`, спецификатор задания можно опустить, если имеется только одно задание.

Возможность перевода в фоновый режим полезна и в том случае, если при запуске программы с графическим интерфейсом из командной строки вы забыли добавить в конец команды символ `&`.

Зачем может понадобиться запускать программу с графическим интерфейсом из командной строки? Тому есть две причины:

- необходимая программа может отсутствовать в меню программ окружения рабочего стола (как, например, `xlogo`);
- запуская программу из командной строки, можно увидеть сообщения об ошибках, которые невидимы, когда программа запускается из графического интерфейса. Иногда программа аварийно завершается при запуске из графического меню. В этом случае, запуская ее из командной строки, можно по сообщениям об ошибках понять причину аварии. Кроме того, некоторые программы с графическим интерфейсом имеют интересные параметры командной строки.

Сигналы

Команда `kill` используется для «убийства» (kill), то есть для завершения процессов. Она позволяет принудительно завершить выполнение вышедшей из-под контроля программы, отвергающей любые другие попытки закрыть ее. Например:

```
[me@linuxbox ~]$ xlogo &
[1] 28401
[me@linuxbox ~]$ kill 28401
[1]+ Terminated              xlogo
```

Здесь мы сначала запустили `xlogo` в фоновом режиме. В ответ командная оболочка вывела номер задания и идентификатор фонового процесса (PID). Затем вызвали команду `kill`, которой передали PID процесса для завершения. Процесс

можно также идентифицировать, указав спецификатор задания (например, %1) вместо PID.

Хотя все выглядит достаточно просто, в действительности команда `kill` не просто «убивает» (kill) процессы — она посылает им *сигналы*. Сигналы — один из нескольких способов, которыми операционная система общается с программами. Мы уже видели сигналы в действии на примере использования комбинаций клавиш CTRL+C и CTRL+Z. Когда терминал принимает одну из этих комбинаций, он посылает сигнал программе на переднем плане. В случае нажатия CTRL+C программе посылается сигнал `INT` (Interrupt — прервать); в случае нажатия CTRL+Z посылается сигнал `TSTP` (Terminal Stop — сигнал «стоп» с клавиатуры). Программы в свою очередь принимают сигналы и могут реагировать на них. Эта возможность позволяет программе выполнить некоторые операции, например, сохранить промежуточные результаты при получении сигнала на завершение.

Посылка сигналов процессам командой kill

Наиболее типичный синтаксис команды `kill` имеет следующий вид:

```
kill [-сигнал] PID...
```

Если сигнал явно не указан в команде, по умолчанию посылается сигнал `TERM` (terminate — завершить). Команда `kill` чаще всего используется для посылки сигналов, перечисленных в табл. 10.4.

Таблица 10.4. Часто используемые сигналы

Номер	Имя	Значение
1	HUP	Обрыв связи. Это пережиток старых добрых времен, когда терминалы подключались к удаленным компьютерам посредством телефонных линий и модемов. Этот сигнал используется, чтобы подсказать программе, что потеряна связь с управляющим терминалом. Действие этого сигнала можно продемонстрировать, закрыв окно терминала. Программа переднего плана, запущенная в терминале, получит сигнал и завершится. Этот сигнал также используется многими программами-демонами для повторной инициализации. То есть когда программа-демон получает этот сигнал, она перезапускается и повторно читает свои конфигурационные файлы. Веб-сервер Apache, например, как раз такая программа-демон, она именно так реагирует на сигнал HUP
2	`INT`	Прервать. Выполняет ту же функцию, что и нажатие комбинации CTRL+C в терминале. Обычно приводит к завершению программы

Номер	Имя	Значение
9	KILL	Уничтожить. Это специальный сигнал. В большинстве случаев программы могут сами решать, как реагировать на сигналы, вплоть до полного их игнорирования, но сигнал KILL в действительности никогда не передается целевой программе. Вместо этого ядро немедленно завершает указанный процесс. Когда процесс завершается таким способом, он не имеет возможности «прибрать за собой» или сохранить результаты своей работы. По этой причине сигнал KILL следует использовать только как крайнее средство, когда другие сигналы на завершение программы не приводят к желаемому результату
15	TERM	Завершить. Это сигнал по умолчанию, посылаемый командой `kill`. Если программа достаточно «живая», чтобы принять этот сигнал, она завершится
18	CONT	Продолжить. Этот сигнал восстанавливает нормальную работу процесса после сигнала STOP
19	STOP	Приостановить. Этот сигнал заставляет процесс приостановиться, не завершаясь. Подобно сигналу KILL, он не передается целевому процессу и потому не может быть проигнорирован им
20	TSTP	Сигнал «стоп» с клавиатуры. Этот сигнал посылается терминалом после нажатия комбинации CTRL+Z. В отличие от сигнала STOP, TSTP передается программе, и программа может решить игнорировать его

Поэкспериментируем с командой `kill`:

```
[me@linuxbox ~]$ xlogo &
[1] 13546
[me@linuxbox ~]$ kill -1 13546
[1]+ Hangup                  xlogo
```

Здесь мы запустили программу `xlogo` в фоновом режиме и затем с помощью команды `kill` послали ей сигнал HUP. Программа `xlogo` завершилась, и командная оболочка сообщила, что фоновый процесс принял сигнал разрыва связи. Иногда необходимо нажать клавишу ENTER пару раз, чтобы увидеть сообщение. Обратите внимание, что сигнал можно указать по номеру или по имени, включая имена сигналов, начинающиеся с префикса *SIG*:

```
[me@linuxbox ~]$ xlogo &
[1] 13601
[me@linuxbox ~]$ kill -INT 13601
[1]+ Interrupt               xlogo
[me@linuxbox ~]$ xlogo &
[1] 13608
[me@linuxbox ~]$ kill -SIGINT 13608
[1]+ Interrupt               xlogo
```

Повторите пример, приведенный выше, и попробуйте послать другие сигналы. Имейте в виду, что вместо PID можно также передавать спецификатор задания.

Процессы, подобно файлам, имеют владельцев, и чтобы послать сигнал процессу командой `kill`, вы должны быть владельцем процесса (или суперпользователем).

Помимо сигналов, наиболее часто используемых с командой `kill` и перечисленных в табл. 10.4, система часто использует другие сигналы, перечисленные в табл. 10.5.

Таблица 10.5. Другие часто используемые сигналы

Номер	Имя	Значение
3	QUIT	Выйти
11	SEGV	Ошибка сегментации. Этот сигнал посылается программе, предпринявшей попытку недопустимого обращения к памяти, то есть попытку выполнить запись в память, доступ к которой запрещен
28	WINCH	Изменение окна. Этот сигнал посылается системой при изменении размеров окна терминала. Некоторые программы, такие как `top` и `less`, реагируют на этот сигнал, обновляя свой вывод в соответствии с новыми размерами окна терминала

Любопытные пользователи могут получить полный список сигналов, выполнив следующую команду:

```
[me@linuxbox ~]$ kill -l
```

Посылка сигналов нескольким процессам с помощью killall

Кроме того, существует возможность с помощью команды `killall` послать сигнал сразу нескольким процессам, соответствующим указанной программе или имени пользователя. Она имеет следующий синтаксис:

```
killall [-u пользователь] [-сигнал] имя...
```

Для демонстрации запустим пару экземпляров программы `xlogo` и затем завершим их:

```
[me@linuxbox ~]$ xlogo &
[1] 18801
[me@linuxbox ~]$ xlogo &
[2] 18802
```

```
[me@linuxbox ~]$ killall xlogo
[1]- Terminated              xlogo
[2]+ Terminated              xlogo
```

Помните: так же как при использовании команды `kill`, вы должны обладать привилегиями суперпользователя, чтобы посылать сигналы процессам, которыми не владеете.

Остановка системы

Для остановки системы перед выключением электропитания требуется по порядку завершить все выполняющиеся процессы, а также выполнить некоторые важные служебные операции (например, синхронизировать все смонтированные файловые системы). Эту задачу решают четыре команды:

- `halt`;
- `poweroff`;
- `reboot`;
- `shutdown`.

Имена первых трех[1] говорят сами за себя. Обычно эти команды вызываются без дополнительных параметров. Например:

```
[me@linuxbox ~]$ sudo reboot
```

Команда `shutdown` немного интереснее. Ей можно указать, какое действие она должна выполнить (остановить, выключить электропитание или перезагрузить), и время задержки перед действием. Чаще она используется для остановки системы:

```
[me@linuxbox ~]$ sudo shutdown -h now
```

и перезагрузки:

```
[me@linuxbox ~]$ sudo shutdown -r now
```

Задержку можно задать разными способами. За подробностями обращайтесь к странице справочного руководства (man). После запуска команда `shutdown` «разошлет» всем зарегистрированным пользователям предупреждение о предстоящем событии.

[1] Их можно перевести как «остановить», «выключить питание» и «перезагрузить» соответственно. — *Примеч. пер.*

Другие команды управления процессами

Так как мониторинг процессов является одной из важнейших задач системного администрирования, существует множество команд, помогающих в этом. В табл. 10.6 перечислены некоторые из них, с ними вы можете поэкспериментировать.

Таблица 10.6. Другие команды управления процессами

Команда	Описание
`pstree`	Выводит список процессов в виде древовидной структуры, отражающей отношения «родитель–потомок» между процессами
`vmstat`	Выводит мгновенный снимок с информацией об использовании системных ресурсов, включая память, файл подкачки и объем дискового ввода/вывода. Чтобы увидеть, как изменяется эта информация с течением времени, передайте команде интервал задержки (в секундах) между обновлениями (например, `vmstat 5`). Завершить работу команды можно нажатием CTRL+C
`xload`	Программа с графическим интерфейсом, показывающая изменение нагрузки на систему с течением времени
`tload`	Работает подобно программе `xload`, но рисует график в терминале. Завершается работа команды нажатием CTRL+C

Заключение

Большинство современных систем обладает механизмом управления процессами. Linux предлагает для этой цели достаточно богатый набор инструментов. Это вполне логично, если учесть, что Linux является самой широко используемой серверной операционной системой в мире. Однако в отличие от некоторых других систем, для управления процессами Linux использует инструменты командной строки. Даже притом, что для Linux имеются инструменты с графическим интерфейсом, инструменты командной строки выглядят предпочтительнее, так как обладают большей скоростью и оказывают меньшую нагрузку на систему. Инструменты с графическим интерфейсом, конечно, выглядят красиво, но сами потребляют довольно много системных ресурсов, что несколько противоречит их назначению.

Часть II
ОКРУЖЕНИЕ И НАСТРОЙКА

11 Окружение

Как говорилось выше, командная оболочка на протяжении всего сеанса работы использует массу информации, которая называется *окружением*. Данные, хранящиеся в окружении, используются программами для выяснения деталей конфигурации системы.

Даже притом, что для хранения своих настроек большинство программ использует *конфигурационные файлы*, некоторые программы также учитывают значения, хранящиеся в окружении. Зная это, можно использовать окружение для настройки некоторых параметров командной оболочки.

В этой главе мы будем работать со следующими командами:

`printenv` — выводит часть или все окружение;

`set` — устанавливает параметры командной оболочки;

`export` — экспортирует окружение для программ, которые будут выполняться позднее;

`alias` — создает псевдоним команды.

Что хранится в окружении?

Командная оболочка хранит в окружении данные двух основных типов, хотя `bash` практически не делает различий между типами. Эти данные хранятся в *переменных окружения* и в *переменных командной оболочки*. Переменные командной оболочки — это фрагменты данных, инициализируемые командой `bash`, а переменные окружения — практически все остальное. Помимо переменных командная

оболочка хранит также программируемые данные, а именно *псевдонимы* и *функции командной оболочки*. Мы уже познакомились с псевдонимами в главе 5, а о функциях (которые имеют отношение к сценариям командной оболочки) поговорим в части IV.

Исследование окружения

Увидеть, что хранится в окружении, можно при помощи встроенной в `bash` команды `set` или программы `printenv`. Команда `set` выводит переменные обоих видов — командной оболочки и окружения, — тогда как `printenv` выводит только последние. Так как список содержимого окружения очень велик, его лучше просматривать, передавая вывод любой из команд по конвейеру в `less`:

```
[me@linuxbox ~]$ printenv | less
```

Запустив эту команду, вы должны увидеть нечто похожее:

```
USER=me
PAGER=less
LSCOLORS=Gxfxcxdxbxegedabagacad
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/usr/share/upstart/xdg:/etc/xdg
PATH=/home/me/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
bin:/usr/games:/usr/local/games
DESKTOP_SESSION=ubuntu
QT_IM_MODULE=ibus
QT_QPA_PLATFORMTHEME=appmenu-qt5
JOB=dbus
PWD=/home/me
XMODIFIERS=@im=ibus
GNOME_KEYRING_PID=1850
LANG=en_US.UTF-8
GDM_LANG=en_US
MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path
MASTER_HOST=linuxbox
IM_CONFIG_PHASE=1
COMPIZ_CONFIG_PROFILE=ubuntu
GDMSESSION=ubuntu
SESSIONTYPE=gnome-session
XDG_SEAT=seat0
HOME=/home/me
SHLVL=2
LANGUAGE=en_US
The Environment 117
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
LESS=-R
```

```
LOGNAME=me
COMPIZ_BIN_PATH=/usr/bin/
LC_CTYPE=en_US.UTF-8
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/share/gnome:/usr/local/share/:/usr/share/
QT4_IM_MODULE=xim
DBUS_SESSION_BUS_ADDRESS=unix:abstract=/tmp/dbus-IwaesmWaT0
LESSOPEN=| /usr/bin/lesspipe %s
INSTANCE=
```

Это список переменных окружения с их значениями. Например, в списке можно увидеть переменную с именем `USER`, содержащую значение `me`. Команда `printenv` может также вывести значение конкретной переменной:

```
[me@linuxbox ~]$ printenv USER
me
```

Команда `set` при вызове без параметров и аргументов выводит переменные обоих типов — командной оболочки и окружения, — а также все объявленные функции командной оболочки.

```
[me@linuxbox ~]$ set | less
```

Получить значение единственной переменной можно также с помощью команды `echo`, например:

```
[me@linuxbox ~]$ echo $HOME
/home/me
```

Единственный элемент окружения, который не выводится командами `set` и `printenv`, это псевдонимы. Чтобы вывести список псевдонимов, используйте команду `alias` без аргументов:

```
[me@linuxbox ~]$ alias
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
alias vi='vim'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-
tilde'
```

Некоторые интересные переменные

Окружение содержит довольно много переменных, и хотя ваше окружение может отличаться от представленного здесь, вы почти наверняка увидите у себя переменные, перечисленные в табл. 11.1.

Таблица 11.1. Переменные окружения

Переменная	Содержит
DISPLAY	Имя вашего дисплея, если вы работаете в графическом окружении. Обычно это `:0`, что означает первый дисплей, сгенерированный X сервером
EDITOR	Имя программы, используемой в качестве текстового редактора
SHELL	Имя программы командной оболочки
HOME	Путь к домашнему каталогу
LANG	Определяет набор символов и порядок сортировки для вашего языка
OLDPWD	Предыдущий рабочий каталог
PAGER	Имя программы для постраничного просмотра. Часто имеет значение */usr/bin/less*
PATH	Список каталогов, разделенных двоеточием, в которых производится поиск выполняемых программ по их именам
PS1	Строка приглашения к вводу №1. Определяет содержимое строки приглашения к вводу в командной оболочке. Как будет показано позднее, эту строку можно менять весьма существенно
PWD	Текущий рабочий каталог
TERM	Тип терминала. Unix-подобные системы поддерживают множество протоколов для работы с терминалами; эта переменная определяет протокол, который будет использоваться при обмене данными с эмулятором терминала
TZ	Определяет часовой пояс. В большинстве Unix-подобных систем внутренние часы компьютера устанавливаются в координированное универсальное время (Coordinated Universal Time, UTC), а при выводе значения времени к нему добавляется смещение, определяемое этой переменной
USER	Ваше имя пользователя

Не волнуйтесь, если какие-то переменные у вас отсутствуют. Они могут отличаться в разных дистрибутивах.

Как устанавливается окружение?

Когда мы входим в систему, запускается программа `bash` и читает содержимое серии конфигурационных сценариев, называемых *файлами запуска* (startup files),

где определяется окружение по умолчанию, общее для всех пользователей. Затем она читает дополнительные файлы запуска в вашем домашнем каталоге, где определяется личное окружение. Точная последовательность обработки файлов зависит от типа запускаемого сеанса командной оболочки. Сеансы работы с командной оболочкой входа могут быть двух типов.

Сеанс командной оболочки входа (login shell session) — это сеанс, который на входе запрашивает имя пользователя и пароль, например, когда вход выполняется в виртуальной консоли.

Сеанс простой командной оболочки (non-login shell session) обычно начинается, когда запускается терминал в графическом окружении.

Командные оболочки входа читают один или несколько файлов запуска, перечисленных в табл. 11.2.

Таблица 11.2. Файлы запуска для сеансов командной оболочки входа

Файл	Содержит
/etc/profile	Общесистемный конфигурационный сценарий, настройки из которого применяются для всех пользователей
~/.bash_profile	Личный пользовательский файл запуска. Может использоваться для расширения и/или переопределения общесистемных настроек
~/.bash_login	Если файл *~/.bash_profile* присутствует в домашнем каталоге, `bash` пытается прочитать его
~/.profile	Если в домашнем каталоге нет ни *~/.bash_profile*, ни *~/.bash_login*, `bash` пытается прочитать этот файл. Используется по умолчанию в дистрибутивах на основе Debian, таких как Ubuntu

Обычные сеансы командной оболочки читают файлы, перечисленные в табл. 11.3.

Таблица 11.3. Файлы запуска для обычных сеансов командной оболочки

Файл	Содержит
/etc/bash.bashrc	Общесистемный конфигурационный сценарий, настройки из которого применяются для всех пользователей
~/.bashrc	Личный пользовательский файл запуска. Может использоваться для расширения и/или переопределения общесистемных настроек

Помимо чтения настроек из файлов запуска, перечисленных в табл. 11.3, обычные командные оболочки наследуют окружение от родительского процесса, каковым обычно является командная оболочка входа.

Загляните в свою систему и посмотрите, какие файлы запуска у вас имеются. Помните: поскольку большинство имен файлов из перечисленных выше начинается с точки (такие файлы считаются скрытыми), при использовании команды `ls` ей необходимо передавать параметр `-a`.

С точки зрения обычного пользователя, файл *~/.bashrc* является, пожалуй, самым важным файлом запуска, потому что его содержимое читается практически всегда. Обычные командные оболочки читают его по умолчанию, а большинство файлов запуска для командных оболочек входа написаны так, что оболочка также прочитает файл *~/.bashrc*.

Что находится в файлах запуска?

Если заглянуть внутрь типичного файла *.bash_profile* (взятого из системы CentOS 6), можно увидеть следующее:

```
# .bash_profile

# Загрузить псевдонимы и функции
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# Определение пользовательского окружения и запуск программ

PATH=$PATH:$HOME/bin
export PATH
```

Строки, начинающиеся с `#`, — это *комментарии*, они не читаются командной оболочкой, а предназначены для человека. Первый интересный фрагмент начинается в четвертой строке:

```
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi
```

Этот код называется *составной условной командой*, полное описание которой будет дано в части IV, где рассматривается программирование на языке командной оболочки. А пока приведем ее перевод на человеческий язык:

```
Если файл "~/.bashrc" существует, тогда
    прочитать файл "~/.bashrc" file.
```

Как видите, этот фрагмент вынуждает командную оболочку входа прочитать содержимое файла *.bashrc*. Следующая операция, выполняемая в файле запуска, имеет отношение к переменной `PATH`.

Приходилось ли вам задумываться над тем, как командная оболочка находит команды, которые вводятся в командной строке? Например, когда мы вводим `ls`, командная оболочка не обыскивает весь компьютер целиком, чтобы найти `/bin/ls` (полный путь к команде `ls`), а просматривает только каталоги, перечисленные в переменной `PATH`.

Переменная `PATH` часто (но не всегда, в зависимости от дистрибутива) устанавливается в файле запуска */etc/profile*, как показано ниже:

```
PATH=$PATH:$HOME/bin
```

Здесь в конец списка в переменной `PATH` добавляется каталог *$HOME/bin*. Этот код может служить примером использования механизма *подстановки параметров*, с которым мы познакомились в главе 7. Для демонстрации попробуйте выполнить следующий пример:

```
[me@linuxbox ~]$ foo="This is some"
[me@linuxbox ~]$ echo $foo
This is some
[me@linuxbox ~]$ foo=$foo" text."
[me@linuxbox ~]$ echo $foo
This is some text.
```

Используя этот прием, можно добавлять текст в конец содержимого переменной.

При добавлении строки `$HOME/bin` в конец содержимого переменной `PATH` происходит добавление каталога *$HOME/bin* в список каталогов, где будет выполняться поиск вводимых команд. Это означает, что если мы решим создать каталог в своем домашнем каталоге для хранения личных программ, командная оболочка уже будет готова к этому. Нам останется только дать имя *bin* этому каталогу.

ПРИМЕЧАНИЕ

Многие дистрибутивы предоставляют настройки `PATH` по умолчанию. Некоторые дистрибутивы на основе Debian, такие как Ubuntu, проверяют наличие каталога *~/bin* во время входа, и если он имеется, динамически добавляют его в переменную `PATH`.

Наконец, у нас осталась еще одна строка:

```
export PATH
```

Команда `export` указывает командной оболочке сделать содержимое переменной `PATH` доступным дочерним процессам этой оболочки.

Изменение окружения

Теперь, зная, где находятся файлы запуска и что они содержат, мы можем изменить их, чтобы настроить окружение.

Какие файлы следует изменять?

Как правило, изменение содержимого переменой `PATH` или определение дополнительных переменных окружения следует производить в файле *.bash_profile* (или эквивалентном ему, в зависимости от дистрибутива, — например, в Ubuntu используется файл *.profile*). Во всех остальных случаях изменения должны производиться в *.bashrc*.

ПРИМЕЧАНИЕ

Если вы не системный администратор и вам не требуется вносить изменения, касающиеся всех пользователей системы, изменяйте только файлы в своем домашнем каталоге. Конечно, можно изменять файлы в */etc*, такие как *profile*, и во многих случаях в этом есть определенный смысл, но давайте пока избегать рискованных действий.

Текстовые редакторы

Для редактирования (то есть изменения) файлов запуска командной оболочки, а также большинства других конфигурационных файлов в системе используется программа, которая называется *текстовым редактором*. Текстовый редактор, подобно текстовому процессору, позволяет редактировать слова на экране, перемещая курсор. От текстового процессора эта программа отличается только поддержкой простого текста и нередко наличием особенностей, необходимых при разработке программ. Текстовые редакторы — основной инструмент, используемый программистами для создания программного кода и системными администраторами для управления конфигурационными файлами, определяющими настройки системы.

Для Linux существует огромное число текстовых редакторов; в вашей системе почти наверняка установлено несколько из них. Почему было создано так много редакторов? Вероятно, потому, что программистам нравится писать их, а так как программисты очень активно пользуются редакторами, они стремятся воплотить в них свои взгляды на то, как должны работать эти редакторы.

Текстовые редакторы делятся на две основные категории: с графическим и с текстовым интерфейсом. Оба окружения рабочего стола, GNOME и KDE, включают несколько популярных редакторов с графическим интерфейсом. В состав GNOME входит редактор с названием `gedit`, который в меню GNOME обычно называется Text Editor (Текстовый редактор). Вместе с KDE обычно распространяется три редактора (в порядке увеличения сложности): `kedit`, `kwrite` и `kate`.

Существует множество редакторов с текстовым интерфейсом. Наиболее популярные из них, с которыми, возможно, вы столкнетесь: `nano`, `vi` и `emacs`. Редактор `nano` — простой в использовании редактор, созданный как замена редактору `pico`, поставляемому в составе пакета программ для работы с электронной почтой PINE. Редактор `vi` (в большинстве систем Linux его замещает программа `vim`, название которой является сокращением от Vi IMproved (Vi улучшенный)) — традиционный редактор для Unix-подобных систем. Подробнее о нем рассказывается в главе 12. Редактор `emacs` был написан Ричардом Столлманом (Richard Stallman). Это невероятная, универсальная среда программирования, построенная по принципу «все в одном». Но, несмотря на свою доступность, он редко устанавливается по умолчанию в большинстве систем Linux.

Использование текстового редактора

Любой текстовый редактор можно запустить из командной строки, введя имя редактора и имя файла, который требуется отредактировать. Если указанный файл не существует, редактор решит, что вы хотите создать новый файл. Ниже приводится пример использования `gedit`:

```
[me@linuxbox ~]$ gedit some_file
```

Эта команда запустит текстовый редактор `gedit` и загрузит в него файл с именем *some_file*, если таковой существует.

Все текстовые редакторы с графическим интерфейсом имеют интуитивно понятный интерфейс, поэтому мы не будем описывать их здесь. Вместо этого сосредоточимся на редакторе с текстовым интерфейсом `nano`. Давайте запустим `nano` и внесем изменения в файл *.bashrc*. Но перед этим поговорим немного о мерах предосторожности. Всякий раз, собираясь редактировать важный конфигурационный файл, создайте сначала его резервную копию. Это обезопасит вас, если

в процессе редактирования вы безнадежно испортите содержимое файла. Чтобы создать резервную копию файла *.bashrc*, выполните следующую команду:

```
[me@linuxbox ~]$ cp .bashrc .bashrc.bak
```

Неважно, как вы назовете файл с резервной копией; просто дайте ему такое имя, чтобы было понятно, что это за файл. Наиболее часто для имен файлов с резервными копиями используются расширения *.bak*, *.sav*, *.old* и *.orig*. Да, и не забудьте, что команда cp без лишних вопросов *затирает существующие файлы*.

Теперь, когда резервная копия создана, можно запускать редактор:

```
[me@linuxbox ~]$ nano .bashrc
```

После запуска nano вы увидите на экране примерно такую картину:

```
  GNU nano 2.0.3         File: .bashrc

# .bashrc

# Загрузить глобальные определения
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi

# Пользовательские псевдонимы и функции


                           [ Read 8 lines ]
^G Get Help^O WriteOut^R Read Fil^Y Prev Pag^K Cut Text^C Cur Pos
^X Exit    ^J Justify ^W Where Is^V Next Pag^U UnCut Te^T To Spell
```

ПРИМЕЧАНИЕ

Если в вашей системе не установлен редактор nano, можете вместо него использовать редактор с графическим интерфейсом.

Экран редактора делится на три части: заголовок в верхней части, область редактирования текста в середине и меню команд внизу. Так как nano проектировался как замена текстового редактора, входящего в состав почтового клиента, он не обладает развитыми функциями редактирования.

Первая команда, которую нужно узнать при использовании любого редактора, — это команда выхода из программы. Чтобы покинуть nano, нажмите CTRL+X.

Эта команда присутствует в меню, в нижней части экрана. Нотация ^X означает CTRL+X. Это распространенная форма записи управляющих комбинаций, используемая во многих программах.

Вторая команда, которую следует знать, — как сохранить изменения. В nano сохранение выполняется нажатием CTRL+O. Теперь, обладая новыми знаниями, приступим к правке текста. Используя клавишу со стрелкой вниз и/или Page Down, переместите курсор в конец файла и добавьте в *.bashrc* следующие строки:

```
umask 0002
export HISTCONTROL=ignoredups
export HISTSIZE=1000
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
```

ПРИМЕЧАНИЕ

Ваш дистрибутив уже может включать некоторые из этих строк, но если повторить их, это ничему не навредит.

Эти изменения описаны в табл. 11.4.

Таблица 11.4. Дополнения в файле .bashrc

Строка	Значение
umask 0002	Определяет маску umask для устранения проблемы с общими каталогами, рассматривавшимися в главе 9
export HISTCONTROL=ignoredups	Предписывает механизму истории командной оболочки игнорировать команду, если непосредственно перед ней точно такая же команда была записана в историю
export HISTSIZE=1000	Увеличивает историю команд с 500 строк по умолчанию до 1000
alias l.='ls -d .* --color=auto'	Создает новую команду с именем l., которая выводит все элементы каталога с именами, начинающимися с точки
alias ll='ls -l –color=auto'	Создает новую команду с именем ll, которая выводит содержимое каталога в подробном формате

ВАЖНОСТЬ КОММЕНТАРИЕВ

Всякий раз, изменяя конфигурационные файлы, добавляйте краткие комментарии, описывающие эти изменения. Вне всяких сомнений, вы будете помнить назначение своих изменений завтра, но вспомните ли вы об этом через шесть месяцев? Сделайте себе подарок, добавьте несколько комментариев. Кроме того, хорошо бы завести файл журнала и в нем фиксировать произведенные изменения.

Комментарии в сценариях на языке командной оболочки и в файлах запуска начинаются с символа #. В других конфигурационных файлах для этой цели могут использоваться другие символы. Комментарии можно найти в большинстве конфигурационных файлов. Используйте их как руководство.

В конфигурационных файлах вам часто будут встречаться *закомментированные* строки, чтобы предотвратить их влияние на соответствующую программу. Это делается с целью показать читателю возможные варианты настройки или примеры правильного синтаксиса оформления настроек. Например, файл *.bashrc* в Ubuntu 18.04 содержит следующие строки:

```
# несколько других псевдонимов команды ls
#alias ll='ls -l'
#alias la='ls -A'
#alias l='ls -CF'
```

Последние три строки — это допустимые определения псевдонимов, только закомментированные. Если удалить начальные символы # из этих трех строк (это называется *раскомментировать*), псевдонимы будут активированы. Напротив, если добавить символ # в начало строки, можно деактивировать конфигурационную строку, сохранив информацию, хранящуюся в ней.

Как видите, догадаться о назначении многих новых строк непросто, поэтому нелишним будет снабдить их комментариями, чтобы прояснить смысл для тех, кто будет читать файл *.bashrc*. Используя редактор, добавьте пояснения, как показано ниже:

```
# Изменить маску umask, чтобы упростить использование общих каталогов
umask 0002

# Игнорировать дубликаты в истории команд и увеличить
# объем истории до 1000 строк
144 Глава 11. Окружение
export HISTCONTROL=ignoredups
export HISTSIZE=1000

# Добавить несколько удобных псевдонимов
```

```
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
```

Так намного лучше! Закончив правку, нажмите CTRL+O, чтобы сохранить измененный файл *.bashrc*, и CTRL+X, чтобы выйти из nano.

Активация изменений

Изменения, произведенные в файле *.bashrc*, не вступят в силу, пока вы не закроете терминал и не запустите новый, потому что оболочка читает содержимое файла *.bashrc* только в начале сеанса. Однако существует возможность принудить bash повторно прочитать измененный файл *.bashrc* следующей командой:

```
[me@linuxbox ~]$ source .bashrc
```

После этого изменения должны вступить в силу. Попробуйте, например, один из новых псевдонимов:

```
[me@linuxbox ~]$ ll
```

Заключение

В этой главе вы приобрели основные навыки правки конфигурационных файлов в текстовом редакторе. По мере чтения страниц справочного руководства (man) для команд обращайте внимание на переменные окружения, поддерживаемые командами. Эта информация может оказаться весьма ценной. В следующих главах вы познакомитесь с еще одним мощным инструментом — функциями командной оболочки, которые также можно включать в файлы запуска bash, чтобы расширить арсенал собственных команд.

12 Плавное введение в vi

Существует старая шутка о человеке, впервые приехавшем в Нью-Йорк и спрашивающем у прохожего дорогу к известному концертному залу:

> Приезжий: Простите, как попасть в Карнеги-холл?
>
> Прохожий: Репетировать, репетировать и репетировать!

Освоение командной строки Linux, как становление пианиста-виртуоза, невозможно за один день. Для этого требуются годы практики. В этой главе вы познакомитесь с текстовым редактором `vi` (произносится как «ви ай»), одной из традиционных программ Unix. Редактор `vi` известен своим сложным пользовательским интерфейсом, но когда вы увидите, как мастер садится за клавиатуру и начинает «играть», вы станете свидетелем высокого искусства. Вы не станете мастерами, прочитав эту главу, но закончив ее, вы будет знать, как сыграть «Собачий вальс» на `vi`.

Зачем осваивать vi

Зачем в современном мире редакторов с графическим интерфейсом и простых в использовании редакторов с текстовым интерфейсом, таких как `nano`, осваивать `vi`? На то есть три веские причины:

- `vi` всегда под рукой. Он может прийти на помощь в системах, где отсутствует графический интерфейс, например, на удаленном сервере или в локальной системе с нерабочей конфигурацией X. Редактор `nano`, хотя и чрезвычайно популярен, все же недостаточно универсален. POSIX, стандарт программной совместимости систем Unix, требует наличия в них `vi`.

- `vi` легковесный и быстрый. Для многих задач гораздо проще запустить `vi`, чем найти в меню редактор с графическим интерфейсом и ждать, пока несколько мегабайтов загрузится в память. Кроме того, `vi` специально проектировался для скоростного ввода с клавиатуры. Как будет показано ниже, опытный пользователь `vi` никогда не отрывает рук от клавиатуры во время редактирования.
- Мы не хотим, чтобы другие пользователи Linux и Unix считали нас неженками.

Хорошо, пусть будет две веские причины.

Немного предыстории

Первая версия `vi` была написана Билли Джоем, студентом Калифорнийского университета в городе Беркли, который позднее стал сооснователем Sun Microsystems в 1976 году. Название `vi` произошло от слова *visual* (экранный), потому что редактор предназначался для редактирования на экране видеотерминала с возможностью перемещения курсора по всей его поверхности. До экранных редакторов существовали строчные редакторы, позволяющие редактировать текст только по одной строке. Чтобы внести изменения в строчном редакторе, нужно было сначала перейти к требуемой строке, а затем описать требуемое изменение: добавление или удаление текста. С появлением видеотерминалов (взамен терминалов с печатающим устройством, таких как телетайпы) стало возможным визуальное редактирование на экране. В действительности `vi` включает в себя мощный строчный редактор `ex`, и его можно использовать для ввода команд во время работы в `vi`.

Большинство дистрибутивов Linux содержат не настоящий редактор `vi`, а его улучшенную замену с именем `vim` (сокращенно от Vi IMproved — Vi улучшенный), созданную Брамом Моленаром (Bram Moolenaar). `vim` намного совершеннее традиционного редактора `vi` и в системах Linux обычно используется под символической ссылкой (или псевдонимом) `vi`. В пояснении ниже будет предполагаться, что у вас есть программа с именем `vi`, которая в действительности является редактором `vim`.

Запуск и завершение vi

Чтобы запустить `vi`, введите следующую команду:

```
[me@linuxbox ~]$ vi
На экране должно появиться:
~
~
~                              VIM - Vi IMproved
~
```

```
~                              version 8.0.707
~                          by Bram Moolenaar et al.
~                Vim is open source and freely distributable
~
~
~                          Sponsor Vim development!
~               type  :help sponsor<Enter>    for information
~
~               type  :q<Enter>               to exit
~               type  :help<Enter>  or  <F1>  for on-line help
~               type  :help version7<Enter>   for version info
~
~                        Running in Vi compatible mode
~               type  :set nocp<Enter>        for Vim defaults
~               type  :help cp-default<Enter> for info on this
~
~
~
```

РЕЖИМ СОВМЕСТИМОСТИ

В примере выше, где показан экран, который выводится сразу после запуска `vim`, можно заметить текст: «Running in Vi compatible mode» (запущен в режиме совместимости с `vi`). Это означает, что `vim` был запущен в режиме, близко повторяющем обычное поведение `vi`, а не в расширенном режиме `vim`. Чтобы беспрепятственно следить за нитью повествования в этой главе, запустите `vim` в расширенном режиме. Для этого в вашем распоряжении имеется пара возможностей. Попробуйте запустить редактор командой `vim` вместо `vi`. Если этот прием сработает, подумайте о том, чтобы добавить псевдоним `vi='vim'` в свой файл *.bashrc*. Или выполните следующую команду, чтобы добавить строку в конфигурационный файл `vim`:

```
echo "set nocp" >> ~/.vimrc
```

В разных дистрибутивах Linux `vim` упакован по-разному. В некоторых по умолчанию устанавливается минимальная версия `vim`, поддерживающая лишь ограниченный набор возможностей. Поэтому, выполняя примеры из этой главы, вы можете столкнуться с отсутствием некоторых возможностей, — в этом случае просто установите полную версию `vim`.

Так же как при знакомстве с nano, которое произошло в предыдущей главе, сначала научимся выходить из редактора. Для этого введите следующую команду (обратите внимание: двоеточие — это часть команды):

```
:q
```

В окне терминала должно появиться приглашение к вводу командной оболочки. Если по какой-то причине выход из `vi` не получился (скорее всего, потому что вы внесли какие-то изменения и еще не сохранили их), сообщите `vi`, что вы действительно хотите выйти, добавив в команду восклицательный знак:

```
:q!
```

СОВЕТ

Если вы «заблудились» в `vi`, попробуйте дважды нажать ESC, чтобы вернуться на верный путь.

Режимы редактирования

Давайте снова запустим `vi`, но на этот раз укажем имя несуществующего файла. Именно так можно с помощью `vi` создать новый файл:

```
[me@linuxbox ~]$ rm -f foo.txt
[me@linuxbox ~]$ vi foo.txt
```

В случае успеха на экране должно появиться следующее:

```
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"foo.txt" [New File]
```

Начальные символы тильды (~) сообщают об отсутствии текста в соответствующих строках. Таким способом vi сообщает нам, что файл пуст. Не вводите пока ничего!

Вторая важная вещь, которую нужно усвоить (после того, как вы научились выходить), — vi является *режимным редактором*. Сразу после запуска vi оказывается в *командном режиме*. В этом режиме практически каждая клавиша является командой, поэтому если вы начнете ввод, vi может запутаться сам и запутать вас.

Переход в режим вставки

Чтобы добавить какой-то текст в файл, необходимо сначала перейти в режим вставки. Для этого нажмите клавишу i. Вслед за этим, если vim работает в обычном расширенном режиме, в нижней части экрана появится надпись (она не появится, если редактор работает в режиме совместимости с vi):

```
-- INSERT --
```

Теперь можно ввести какой-нибудь текст. Попробуйте, например:

```
Съешь же ещё этих мягких французских булок, да выпей чаю.
```

Чтобы выйти из режима вставки и вернуться в командный режим, нажмите ESC.

Сохранение изменений

Чтобы сохранить изменения в файл, введите ex-команду, находясь в командном режиме. Для этого нажмите клавишу :. После этого в нижней части должен появиться символ двоеточие:

```
:
```

Чтобы выполнить запись изменений в файл, вслед за двоеточием введите w и нажмите ENTER:

```
:w
```

Файл будет записан на жесткий диск, и в нижней части появится подтверждение:

```
"foo.txt" [New] 1L, 46C written
```

СОВЕТ

Если заглянуть в документацию к vim, можно заметить, что (по непонятной причине) командный режим в ней называется нормальным режимом, а ex-команды называются командным режимом. Имейте эту неточность в виду.

Перемещение курсора

Находясь в командном режиме, `vi` предлагает большое число команд управления курсором, часть из которых также используется программой `less`. В табл. 12.1 перечислены некоторые из этих команд.

Таблица 12.1. Клавиши управления курсором

Клавиша	Перемещает курсор
`l` или стрелка вправо	Вправо на один символ
`h` или стрелка влево	Влево на один символ
`j` или стрелка вниз	Вниз на одну строку
`k` или стрелка вверх	Вверх на одну строку
`0` (ноль)	В начало текущей строки
`^`	К первому непробельному символу в текущей строке
`$`	В конец текущей строки
`w`	В начало следующего слова или к знаку препинания
`W`	В начало следующего слова, минуя знаки препинания
`b`	В начало предыдущего слова или к знаку препинания
`B`	В начало предыдущего слова, минуя знаки препинания
CTRL+F или Page Down	Вниз на одну страницу
CTRL+B или Page Up	Вверх на одну страницу
число`G`	К строке с указанным номером (например, команда `1G` выполнит переход к первой строке в файле)
`G`	К последней строке в файле

Почему для перемещения курсора были выбраны клавиши `h`, `j`, `k` и `l`? Потому что, когда был написан редактор `vi`, не все видеотерминалы имели кнопки со стрелками на клавиатуре. Таким образом, опытные пользователи, хорошо владеющие клавиатурой, могли управлять курсором, не отрывая пальцев от клавиш.

Многие команды в `vi` могут начинаться с числа, как команда `G` в табл. 12.1. Добавляя число в команду, можно указать, сколько раз она должна быть выполнена. Например, команда `5j` переместит курсор на пять строк вниз.

Основы редактирования

Редактирование в основном заключается в нескольких простых операциях, таких как вставка текста, удаление текста и перемещение фрагментов текста с применением операций вырезания и вставки. Конечно же, `vi` поддерживает все эти операции своим неповторимым способом. `vi` поддерживает ограниченную форму отмены. Если нажать клавишу `u` в командном режиме, `vi` отменит последнее выполненное изменение. Это пригодится нам, когда мы будем пробовать некоторые простые команды редактирования.

Добавление текста в конец

`vi` поддерживает несколько способов входа в режим вставки. Мы уже использовали команду `i` для вставки текста.

Давайте вернемся к нашему файлу *foo.txt*:

```
Съешь же ещё этих мягких французских булок, да выпей чаю.
```

Если попытаться добавить текст в конец приложения, можно обнаружить, что команда `i` не позволяет сделать это, не давая переместить курсор за конец строки. `vi` поддерживает команду добавления текста в конец, разумно названную `a`. Если переместить курсор в конец строки и ввести `a`, курсор переместится за конец строки и `vi` перейдет в режим вставки. Это позволит нам добавить следующий текст:

```
Съешь же ещё этих мягких французских булок, да выпей чаю. Это классно.
```

Не забудьте нажать ESC, чтобы выйти из режима вставки.

Поскольку добавлять текст в конец строки требуется довольно часто, `vi` предлагает сокращенную команду для перемещения в конец строки и перехода в режим добавления. Это команда `A`. Давайте попробуем с ее помощью добавить еще несколько строк в наш файл.

Сначала командой `0` (ноль) переместите курсор в начало строки. Затем введите `A` и добавьте следующие строки текста:

```
Съешь же ещё этих мягких французских булок, да выпей чаю. Это классно.
Строка 2
Строка 3
Строка 4
Строка 5
```

Снова нажмите клавишу ESC, чтобы выйти из режима вставки.

Как видите, команда A очень удобна, потому что помещает курсор в конец строки перед переходом в режим вставки.

Вставка строки

Другой способ вставки текста — вставка строк. Он позволяет вставить пустую строку между двумя имеющимися строками и перейти в режим вставки. Данный способ предлагает два варианта вставки, перечисленные в табл. 12.2.

Таблица 12.2. Команды вставки строк

Команда	Вставляет
o	Строку ниже текущей
O	Строку выше текущей

Рассмотрим действие этих команд на следующих примерах: поместите курсор в строку с текстом *Строка 3* и введите o.

```
Съешь же ещё этих мягких французских булок, да выпей чаю. Это классно.
Строка 2
Строка 3

Строка 4
Строка 5
```

Под третьей строкой появилась пустая строка, и редактор перешел в режим вставки. Выйдите из режима вставки нажатием ESC. Введите u, чтобы отменить изменения. Введите O, чтобы вставить пустую строку выше курсора:

```
Съешь же ещё этих мягких французских булок, да выпей чаю. Это классно.
Строка 2

Строка 3
Строка 4
Строка 5
```

Выйдите из режима вставки нажатием ESC и введите `u`, чтобы отменить изменения.

Удаление текста

Как можно догадаться, `vi` предлагает несколько способов удаления текста, и все они требуют нажатия одной или двух клавиш. Первый способ: команда `x` удаляет символ в позиции курсора. Команде `x` может предшествовать число, определяющее количество удаляемых символов. Команда `d` более универсальна. Команде `d` также может предшествовать число, определяющее количество операций удаления. Кроме того, команда `d` всегда сопровождается командой перемещения курсора, управляющей размером удаляемой области. В табл. 12.3 приводится несколько примеров команды удаления.

Таблица 12.3. Команды удаления текста

Команда	Удалит
`x`	Текущий символ
`3x`	Текущий символ и следующие за ним два символа
`dd`	Текущую строку
`5dd`	Текущую строку и следующие за ней четыре строки
`dW`	От символа в текущей позиции курсора до начала следующего слова
`d$`	От символа в текущей позиции курсора до конца текущей строки
`d0`	От символа в текущей позиции курсора до начала строки
`d^`	От символа в текущей позиции курсора до первого непробельного символа в строке
`dG`	От текущей строки до конца файла
`d20G`	От текущей строки до 20-й строки файла

Поместите курсор на слово *Это* в первой строке. Вводите `x`, пока текст до конца предложения не будет удален. Затем введите несколько раз команду `u`, чтобы отменить удаление.

ПРИМЕЧАНИЕ

Настоящий редактор `vi` поддерживает отмену только самой последней команды. `vim` поддерживает отмену множества команд.

Теперь давайте проведем операцию удаления еще раз, но теперь воспользуемся командой d. Снова установите курсор на слово *Это* и введите dW, чтобы удалить слово:

```
Съешь же ещё этих мягких французских булок, да выпей чаю. классно.
Строка 2
Строка 3
Строка 4
Строка 5
```

Введите d$, чтобы удалить все от текущей позиции курсора до конца строки:

```
Съешь же ещё этих мягких французских булок, да выпей чаю.
Строка 2
Строка 3
Строка 4
Строка 5
```

Введите dG, чтобы удалить все от текущей строки до конца файла:

```
~
~
~
~
~
```

Введите u три раза, чтобы отменить операции удаления.

Вырезание, копирование и вставка текста

Команда d не просто удаляет текст, она «вырезает» его. Каждый раз, когда выполняется команда d, удаленный текст копируется в буфер вставки (своего рода буфер обмена — clipboard), откуда позднее его можно извлечь командой p и вставить правее позиции курсора или левее — командой P.

Команда y выполняет копирование (yank) текста в буфер вставки почти так же, как команда d. В табл. 12.4 перечислены некоторые примеры комбинирования команды y с разными командами перемещения курсора.

Таблица 12.4. Команды копирования текста

Команда	Скопирует
yy	Текущую строку
5yy	Текущую и следующие четыре строки

Команда	Скопирует
yW	От текущей позиции курсора до начала следующего слова
y$	От текущей позиции курсора до конца текущей строки
y0	От текущей позиции курсора до начала строки
y^	От текущей позиции курсора до первого непробельного символа в строке
yG	От текущей строки до конца файла
y20G	От текущей строки до 20-й строки файла

Давайте попробуем что-нибудь скопировать и вставить. Поместите курсор на первую строку и введите yy, чтобы скопировать текущую строку. Далее, переместите курсор в последнюю строку (G) и введите p, чтобы вставить скопированную строку ниже текущей:

```
Съешь же ещё этих мягких французских булок, да выпей чаю. Это классно.
Строка 2
Строка 3
Строка 4
Строка 5
Съешь же ещё этих мягких французских булок, да выпей чаю. Это классно.
```

Введите команду u, чтобы отменить изменение. Оставив курсор в последней строке, введите P, чтобы вставить текст выше текущей строки:

```
Съешь же ещё этих мягких французских булок, да выпей чаю. Это классно.
Строка 2
Строка 3
Строка 4
Съешь же ещё этих мягких французских булок, да выпей чаю. Это классно.
Строка 5
```

Попробуйте другие команды y из табл. 12.4 и посмотрите, как действуют команды p и P. Закончив эксперименты, верните файл в исходное состояние.

Объединение строк

vi очень строго относится к понятию строки. Обычно он не дает возможности переместить курсор в конец строки и удалить символ конца строки, чтобы объединить текущую строку со следующей за ней. По этой причине в vi была добавлена специальная команда J (не путайте с командой j, которая перемещает курсор на одну строку вниз) для объединения строк.

Если поместить курсор в третью строку и ввести команду J, получится следующее:

```
Съешь же ещё этих мягких французских булок, да выпей чаю. Это классно.
Строка 2
Строка 3 Строка 4
Строка 5
```

Поиск и замена

Редактор vi имеет возможность перемещать курсор, опираясь на результаты поиска. Он может это делать в пределах одной строки или всего файла. Он может также выполнять замену текста с подтверждением или без подтверждения пользователя.

Поиск в пределах строки

Команда f выполняет поиск в строке и перемещает курсор к следующему вхождению указанного символа. Например, команда fa переместит курсор к следующему вхождению символа *a* в текущей строке. После выполнения операции поиска символа в строке ее можно повторить, введя точку с запятой.

Поиск во всем файле

Для перемещения курсора к следующему вхождению слова или фразы используется команда /. Она действует точно так же, как в программе less, о которой рассказывалось в главе 3. После ввода команды / в нижней части экрана появится прямой слеш, вслед за которым нужно ввести искомое слово или фразу и нажать ENTER. После этого курсор переместится к следующему вхождению искомой строки. Поиск следующего вхождения той же строки можно повторить командой n. Например:

```
Съешь же ещё этих мягких французских булок, да выпей чаю. Это классно.
Строка 2
Строка 3
Строка 4
Строка 5
```

Поместите курсор в первую строку и введите

```
/Строка
```

затем нажмите ENTER. Курсор переместится в строку 2. Затем введите команду n, и курсор переместится в строку 3. С каждой следующей командой n курсор будет

перемещаться вниз по файлу, пока не достигнет последнего вхождения искомого фрагмента. В примерах выше мы использовали для поиска только слова и фразы, однако `vi` позволяет применять *регулярные выражения* — очень мощное средство выражения сложных шаблонов текста. Мы подробно обсудим регулярные выражения в главе 19.

Глобальный поиск и замена

Для выполнения поиска с заменой (в `vi` эта операция называется *подстановкой*) в диапазоне строк или во всем файле `vi` использует ex-команды. Например, заменить слово *Строка* словом *строка* во всем файле можно следующей командой:

```
:%s/Строка/строка/g
```

Давайте разобьем эту команду на элементы и рассмотрим их по отдельности (табл. 12.5).

Таблица 12.5. Пример синтаксиса команды глобального поиска с заменой

Элемент	Значение
`:`	Символ «двоеточие» начинает ex-команду
`%`	Определяет диапазон строк, где будет выполняться поиск. `%` — сокращение, означающее «от первой строки до последней». В этом примере можно было бы указать диапазон номеров строк `1,5` (потому что в нашем файле всего пять строк) или `1,$`, что означает «от строки с номером 1 до последней». Если диапазон строк не указан, операция применяется только к текущей строке
`s`	Определяет операцию — в данном случае подстановку (substitution), или поиск с заменой
`/Строка/строка/`	Шаблон поиска и текст замены
`g`	Означает *global* (глобально), в том смысле, что подстановка выполняется для всех вхождений искомой строки в каждой строке. Если элемент `g` опустить, операция выполнит замену только первого вхождения искомого фрагмента в каждой строке

После выполнения поиска с заменой наш файл будет выглядеть так:

```
Съешь же ещё этих мягких французских булок, да выпей чаю. Это классно.
строка 2
строка 3
строка 4
строка 5
```

В команде подстановки можно указать, что она должна запрашивать подтверждение у пользователя перед заменой. Для этого добавьте символ `c` в конец команды. Например:

```
:%s/строка/Строка/gc
```

Эта команда вернет содержимое файла в прежнее состояние, но перед каждой заменой `vi` будет останавливаться и спрашивать подтверждение, выдавая следующее сообщение:

```
заменить на Строка? (y/n/a/q/l/^E/^Y)
```

В круглых скобках перечислены возможные варианты ответов, описание которых приводится в табл. 12.6.

Таблица 12.6. Клавиши подтверждения замены

Клавиша	Действие
y	Выполнить замену
n	Пропустить найденное вхождение
a	Выполнить замену этого и всех последующих вхождений
q или ESC	Завершить операцию
l	Выполнить замену этого вхождения и завершить операцию. Сокращенно от *last* (последняя)
CTRL+E, CTRL+Y	Прокрутить вниз или вверх соответственно. Эти команды удобно использовать для просмотра контекста найденного вхождения перед заменой

Редактирование нескольких файлов

Иногда бывает необходимо редактировать сразу несколько файлов. Например, может понадобиться внести изменения в файлы или скопировать содержимое из одного файла в другой. Редактор `vi` позволяет открыть несколько файлов, перечислив их в командной строке:

```
vi файл1 файл2 файл3...
```

Давайте закроем текущий сеанс работы vi и создадим новый файл для редактирования. Введите :wq, чтобы выйти из vi с сохранением изменений в тексте. Далее, создайте новый файл в домашнем каталоге, который мы будем использовать в наших экспериментах. Создайте файл, захватив в него вывод команды ls:

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Теперь откройте в vi старый и новый файлы:

```
[me@linuxbox ~]$ vi foo.txt ls-output.txt
```

После запуска vi вы увидите на экране первый файл:

```
Съешь же ещё этих мягких французских булок, да выпей чаю. Это классно.
Строка 2
Строка 3
Строка 4
Строка 5
```

Переключение между файлами

Чтобы переключиться с одного файла на следующий, выполните ex-команду:

```
:bn
```

Чтобы вернуться обратно, в предыдущий файл, выполните:

```
:bp
```

Теперь мы можем переключаться между файлами, но vi проводит политику, запрещающую переключаться между файлами, если в текущем файле имеются несохраненные изменения. Чтобы заставить vi переключиться между файлами с потерей всех несохраненных изменений, добавьте в команду восклицательный знак (!).

В дополнение к методам переключения между файлами, описанным выше, vim (и некоторые версии vi) предоставляет дополнительные ex-команды, упрощающие управление множеством файлов. Например, командой :buffers можно вывести список редактируемых файлов. В этом случае список появляется в нижней части экрана:

```
:buffers

  1 %a   "foo.txt"           line 1
  2      "ls-output.txt"     line 0
Нажмите ENTER или введите команду для продолжения
```

Чтобы перейти к другому буферу (файлу), введите `:buffer` и номер искомого буфера. Например, переключиться с буфера 1, содержащего файл *foo.txt*, на буфер 2, содержащий файл *ls-output.txt*, можно командой:

```
:buffer 2
```

после выполнения этой команды на экране появится второй файл.

Открытие дополнительных файлов для редактирования

Также существует возможность добавлять файлы в текущий сеанс редактирования. Команда `:e` (сокращенно от edit — редактировать) с именем файла откроет дополнительный файл. Завершите текущий сеанс редактирования и вернитесь в командную строку.

Запустите `vi` снова, но на этот раз с единственным файлом:

```
[me@linuxbox ~]$ vi foo.txt
```

Чтобы добавить второй файл, введите:

```
:e ls-output.txt
```

и на экране должен появиться второй файл. Первый файл останется открытым в редакторе, в чем легко убедиться:

```
:buffers
  1 #       "foo.txt"        line 1
  2 %a      "ls-output.txt"  line 0
Нажмите ENTER или введите команду для продолжения
```

Копирование содержимого из одного файла в другой

Часто в процессе редактирования множества файлов бывает необходимо скопировать фрагмент текста из одного файла в другой. Это легко сделать с помощью обычных команд копирования и вставки, представленных выше. Посмотрим, как можно это осуществить. Сначала, в случае с использованием двух наших файлов, переключитесь на буфер 1 (*foo.txt*), выполнив команду:

```
:buffer 1
```

В результате на экране должно появиться следующее:

```
Съешь же ещё этих мягких французских булок, да выпей чаю. Это классно.
Строка 2
Строка 3
Строка 4
Строка 5
```

Далее переместите курсор на первую строку и введите yy, чтобы скопировать строку.

Переключитесь на второй буфер командой:

```
:buffer 2
```

Теперь на экране должен появиться некий список файлов (здесь показана только часть):

```
total 343700
-rwxr-xr-x 1 root root      31316 2017-12-05 08:58 [
-rwxr-xr-x 1 root root       8240 2017-12-09 13:39 411toppm
-rwxr-xr-x 1 root root     111276 2018-01-31 13:36 a2p
-rwxr-xr-x 1 root root      25368 2017-10-06 20:16 a52dec
-rwxr-xr-x 1 root root      11532 2017-05-04 17:43 aafire
-rwxr-xr-x 1 root root       7292 2017-05-04 17:43 aainfo
```

Переместите курсор на первую строку и вставьте строку, скопированную в предыдущем файле, введя команду p:

```
total 343700
Съешь же ещё этих мягких французских булок, да выпей чаю. Это классно.
-rwxr-xr-x 1 root root      31316 2017-12-05 08:58 [
-rwxr-xr-x 1 root root       8240 2017-12-09 13:39 411toppm
-rwxr-xr-x 1 root root     111276 2018-01-31 13:36 a2p
-rwxr-xr-x 1 root root      25368 2017-10-06 20:16 a52dec
-rwxr-xr-x 1 root root      11532 2017-05-04 17:43 aafire
-rwxr-xr-x 1 root root       7292 2017-05-04 17:43 aainfo
```

Вставка целого файла в другой файл

Кроме того, мы можем вставить файл целиком в другой файл. Для выполнения этого приема завершите сеанс vi и запустите новый, с одним файлом:

```
[me@linuxbox ~]$ vi ls-output.txt
```

На экране снова появится список файлов:

```
total 343700
-rwxr-xr-x 1 root root      31316 2017-12-05 08:58 [
-rwxr-xr-x 1 root root       8240 2017-12-09 13:39 411toppm
-rwxr-xr-x 1 root root     111276 2018-01-31 13:36 a2p
-rwxr-xr-x 1 root root      25368 2017-10-06 20:16 a52dec
-rwxr-xr-x 1 root root      11532 2017-05-04 17:43 aafire
-rwxr-xr-x 1 root root       7292 2017-05-04 17:43 aainfo
```

Переместите курсор в третью строку и введите следующую ex-команду:

```
:r foo.txt
```

Команда `:r` (сокращенно от read — читать) вставит указанный файл перед позицией курсора. Теперь экран должен выглядеть так:

```
total 343700
-rwxr-xr-x 1 root root      31316 2017-12-05 08:58 [
-rwxr-xr-x 1 root root       8240 2017-12-09 13:39 411toppm
Съешь же ещё этих мягких французских булок, да выпей чаю. Это классно.
Строка 2
Строка 3
Строка 4
Строка 5
-rwxr-xr-x 1 root root     111276 2018-01-31 13:36 a2p
-rwxr-xr-x 1 root root      25368 2017-10-06 20:16 a52dec
-rwxr-xr-x 1 root root      11532 2017-05-04 17:43 aafire
-rwxr-xr-x 1 root root       7292 2017-05-04 17:43 aainfo
```

Сохранение результатов работы

И здесь `vi` предлагает нам несколько способов сохранения отредактированных файлов. Мы уже знакомы с ex-командой `:w`, но существуют и другие команды, которые могут оказаться полезными.

В командном режиме можно ввести `ZZ`, чтобы сохранить текущий файл и выйти из `vi`. Аналогично, ex-команда `:wq` сочетает в себе команды `:w` и `:q`, первая из которых сохраняет файл, а вторая закрывает редактор.

В команде `:w` можно также указать имя файла. В этом случае она будет действовать как команда Save As (Сохранить как). Например, если вы редактируете *foo.txt* и хотите сохранить альтернативную версию с именем *foo1.txt*, введите следующую команду:

```
:w foo1.txt
```

ПРИМЕЧАНИЕ

Эта команда сохранит файл с новым именем, но она не изменит имя текущего редактируемого файла. Если после этого продолжить редактирование, вы будете редактировать *foo.txt*, а не *foo1.txt*.

Заключение

Освоив эти основные навыки, вы сможете решать простейшие задачи администрирования типичной системы Linux, связанные с редактированием текста. Время, потраченное на изучение приемов использования `vim`, обязательно окупится в будущем. `vi`-подобные редакторы глубоко укоренились в культуре Unix, и мы увидим множество других программ, подвергшихся их влиянию. Ярким примером такого влияния может служить программа `less`.

13 Настройка приглашения к вводу

В этой главе мы рассмотрим, казалось бы, такую незначительную деталь, как приглашение к вводу командной оболочки (prompt). Кроме того, мы познакомимся с некоторыми внутренними особенностями работы командной оболочки и самой программы эмулятора терминала.

Как и многое в Linux, приглашение к вводу командной оболочки можно настраивать в очень широких пределах, и хотя мы принимаем это приглашение как данность, оно в действительности оказывается очень полезным средством для тех, кто умеет управлять им.

Устройство строки приглашения к вводу

По умолчанию строка приглашения к вводу имеет следующий вид:

```
[me@linuxbox ~]$
```

Обратите внимание, что она содержит имя пользователя, имя хоста (сетевое имя компьютера) и имя текущего рабочего каталога. Но как она образовалась? Все, оказывается, очень просто. Форма приглашения к вводу определяется в переменной окружения `PS1` (сокращенно от *prompt string 1* — строка приглашения 1). Увидеть содержимое переменной `PS1` можно с помощью команды `echo`:

```
[me@linuxbox ~]$ echo $PS1
[\u@\h \W]\$
```

ПРИМЕЧАНИЕ

Не волнуйтесь, если вы увидите нечто отличающееся от примера, приведенного выше. Все дистрибутивы Linux определяют приглашение к вводу по-своему, а некоторые содержат весьма экзотические определения.

Мы видим, что `PS1` содержит несколько символов, например: квадратные скобки, знак `@` и знак доллара, но все остальное — сплошная абракадабра. Наиболее догадливые из вас сразу поймут, что *символы, экранированные слешем*, — *специальные символы*, как те, что мы видели в главе 7. В табл. 13.1 приводится неполный список символов, которые командная оболочка интерпретирует специальным образом в строке приглашения.

Таблица 13.1. Экранированные последовательности, используемые в строке приглашения

Последовательность	Отображаемое значение
`\a`	Звонок. Заставляет компьютер издавать звуковой сигнал
`\d`	Текущая дата в формате: день недели месяц число; например, «Mon May 26» (Пн Май 26)
`\h`	Имя хоста локальной машины минус имя домена
`\H`	Полное имя хоста
`\j`	Число заданий, действующих в текущем сеансе
`\l`	Имя текущего устройства терминала
`\n`	Символ перевода строки
`\r`	Возврат каретки
`\s`	Имя программы командной оболочки
`\t`	Текущее время в 24-часовом формате
`\T`	Текущее время в 12-часовом формате
`\@`	Текущее время в 12-часовом формате `AM/PM`
`\A`	Текущее время в 24-часовом формате, часы:минуты

Таблица 13.1 (*окончание*)

Последовательность	Отображаемое значение
\u	Имя пользователя
\v	Номер версии командной оболочки
\V	Номер версии и выпуска командной оболочки
\w	Имя текущего рабочего каталога
\W	Последняя часть в имени текущего рабочего каталога
\!	Номер текущей команды в истории
\#	Число команд, введенных в текущем сеансе командной оболочки
\$	Выводит символ $, если пользователь не является суперпользователем, иначе выводит символ #
\[	Отмечает начало последовательности из одного или более непечатаемых символов. Используется для внедрения непечатаемых символов, управляющих поведением эмулятора терминала, например, перемещает курсор или изменяет цвет текста
\]	Отмечает конец последовательности непечатаемых символов

Альтернативные варианты оформления приглашения

Имея список специальных символов, можно попробовать изменить оформление приглашения. Для начала сохраните исходное определение, чтобы его можно было восстановить позднее. Для этого скопируйте значение переменной `PS1` в другую переменную:

```
[me@linuxbox ~]$ ps1_old="$PS1"
```

Здесь создается новая переменная с именем `ps1_old`, и ей присваивается значение переменной `PS1`. Убедиться, что значение скопировано, можно с помощью команды `echo`:

```
[me@linuxbox ~]$ echo $ps1_old
[\u@\h \W]\$
```

Это позволит вам в любой момент восстановить исходное оформление приглашения, выполнив обратную процедуру:

```
[me@linuxbox ~]$ PS1="$ps1_old"
```

Теперь, когда все готово, давайте посмотрим, что получится, если определить пустую строку приглашения:

```
[me@linuxbox ~]$ PS1=
```

Если определить приглашение как пустую строку, мы ничего не увидим. Строка приглашения просто исчезнет! В действительности она все еще существует, но поскольку она пустая, на экране ничего не отображается, — собственно, как мы и просили. Так как пустая строка приглашения дезориентирует, давайте определим минимальное оформление:

```
PS1="\$ "
```

Так лучше. По крайней мере, теперь видно, где мы находимся. Обратите внимание на завершающий пробел внутри кавычек. Он обеспечивает дополнительное пространство на экране между знаком доллара и курсором.

Добавим в строку приглашения сигнал:

```
$ PS1="\[\a\]\$ "
```

Теперь при каждом выводе строки приглашения вы должны слышать короткий звуковой сигнал. Постоянно звучащий сигнал может раздражать, но в некоторых случаях он может быть полезен, например, если нужно получать звуковое оповещение об удачном завершении долго выполняющихся команд. Обратите внимание, что мы добавили последовательности `\[` и `\]`. Поскольку управляющий ASCII-символ «звонок» (`\a`) не является «печатаемым», то есть не перемещает курсор при выводе, мы должны сообщить об этом командной оболочке `bash`, чтобы та могла правильно определить длину строки приглашения.

А теперь попробуйте сделать приглашение более информативным, добавив имя хоста и время суток:

```
$ PS1="\A \h \$ "
17:33 linuxbox $
```

Добавление времени суток может пригодиться, если есть необходимость зафиксировать, в какой момент закончилось выполнение задачи. Наконец, сделайте приглашение похожим на оригинальное:

```
17:37 linuxbox $ PS1="<\u@\h \W>\$ "
<me@linuxbox ~>$
```

Попробуйте использовать другие последовательности из табл. 13.1 и посмотрите, сможете ли вы получить свою уникальную строку приглашения к вводу.

Добавление цвета

Большинство программ эмуляторов терминалов реагируют на некоторые последовательности непечатаемых символов, например, управляющие атрибутами символов (такими, как цвет, жирность и мигание) и позицией курсора. О позиции курсора мы поговорим чуть позже, а сейчас займемся цветом.

НЕДОРАЗУМЕНИЯ С ТЕРМИНАЛАМИ

В стародавние времена, когда дискеты были большими, а терминалы подключались к удаленным компьютерам, существовало великое многообразие моделей терминалов, и все они работали по-разному. Они имели разные клавиатуры и по-разному интерпретировали управляющую информацию. В Unix и в Unix-подобных системах имеются две очень сложные подсистемы (которые называются `termcap` и `terminfo`), решающие все проблемы, связанные с управлением терминалами. Если заглянуть в самые потаенные кладовые настроек эмулятора терминала, можно обнаружить параметр, определяющий тип эмулируемого терминала.

Чтобы заставить терминалы говорить на едином языке, Американский национальный институт стандартов (American National Standards Institute, ANSI) разработал набор последовательностей символов для управления видеотерминалами. Заслуженные пользователи DOS еще помнят файл *ANSI.SYS*, который применялся для интерпретации этих последовательностей.

Цветом символов можно управлять, посылая эмулятору терминала *экранированные последовательности ANSI* внутри потока символов, предназначенных для вывода на экран. Экранированные последовательности не «выводятся» на экран; они интерпретируются терминалом как инструкции. Как показано в табл. 13.1, для включения непечатаемых символов используются последовательности `\[` и `\]`. Экранированные последовательности ANSI начинаются с восьмеричного кода 033 (код, генерируемый клавишей ESC), за которым следует необязательный атрибут символа и инструкция. Например, вот как выглядит код, определяющий текст как простой (атрибут = 0), черного цвета

```
\033[0;30m.
```

В табл. 13.2 перечислены поддерживаемые цвета текста. Обратите внимание, что цвета делятся на две группы, отличаясь наличием атрибута жирного текста (1), из-за которого возникает впечатление более «светлых» (light) цветов.

Таблица 13.2. Экранированные последовательности, используемые для определения цвета текста

Последовательность	Цвет	Последовательность	Цвет
`\033[0;30m`	Черный	`\033[1;30m`	Темно-серый
`\033[0;31m`	Красный	`\033[1;31m`	Светло-красный
`\033[0;32m`	Зеленый	`\033[1;32m`	Светло-зеленый
`\033[0;33m`	Коричневый	`\033[1;33m`	Желтый
`\033[0;34m`	Синий	`\033[1;34m`	Светло-синий
`\033[0;35m`	Пурпурный	`\033[1;35m`	Светло-пурпурный
`\033[0;36m`	Бирюзовый	`\033[1;36m`	Светло-бирюзовый
`\033[0;37m`	Светло-серый	`\033[1;37m`	Белый

Давайте попробуем окрасить строку приглашения в красный цвет (здесь она выглядит как серая). Добавьте в начало экранированную последовательность:

```
<me@linuxbox ~>$ PS1="\[\033[0;31m\]<\u@\h \W>\$ "
<me@linuxbox ~>$
```

Получилось, но обратите внимание, что весь текст, который вводится с клавиатуры вслед за строкой приглашения, также окрашивается в красный цвет. Для устранения этого эффекта добавьте еще одну экранированную последовательность в конец определения приглашения — этим вы сообщите эмулятору терминала, что тот должен восстановить нормальный цвет:

```
<me@linuxbox ~>$ PS1="\[\033[0;31m\]<\u@\h \W>\$\[\033[0m\] "
<me@linuxbox ~>$
```

Так лучше!

Кроме того, существует возможность изменить цвет фона, для чего предназначены экранированные последовательности, перечисленные в табл. 13.3. Цвет фона не поддерживает атрибут жирного текста.

Таблица 13.3. Экранированные последовательности, используемые для определения цвета фона

Последовательность	Цвет	Последовательность	Цвет
\033[0;40m	Черный	\033[0;44m	Синий
\033[0;41m	Красный	\033[0;45m	Пурпурный
\033[0;42m	Зеленый	\033[0;46m	Бирюзовый
\033[0;43m	Коричневый	\033[0;47m	Светло-серый

Чтобы вывести приглашение на красном фоне, достаточно изменить первую экранированную последовательность:

```
<me@linuxbox ~>$ PS1="\[\033[0;41m\]<\u@\h \W>\$\[\033[0m\] "
<me@linuxbox ~>$
```

Попробуйте другие цвета и посмотрите, что из этого получится!

ПРИМЕЧАНИЕ

Помимо атрибутов символов нормального (0) и жирного (1) текста, есть также атрибут подчеркивания (4), мигания (5) и инверсии (7). В целях воспитания хорошего вкуса многие эмуляторы терминалов не поддерживают атрибут мигания.

Перемещение курсора

Экранированные последовательности можно использовать для позиционирования курсора. Этот прием часто используется для отображения времени или другой информации в разных местах на экране, например вверху, при каждом выводе приглашения к вводу. В табл. 13.4 перечислены экранированные последовательности, управляющие позицией курсора.

Таблица 13.4. Экранированные последовательности, управляющие позицией курсора

Последовательность	Действие
\033[*l*;*c*H	Перемещает курсор в строку *l* и позицию *c*
\033[*n*A	Перемещает курсор вверх на *n* строк
\033[*n*B	Перемещает курсор вниз на *n* строк
\033[*n*C	Перемещает курсор вперед на *n* символов

Последовательность	Действие
\033[*n*D	Перемещает курсор назад на *n* символов
\033[2J	Очищает экран и помещает курсор в левый верхний угол (строка 0, позиция 0)
\033[K	Очищает экран от позиции курсора до конца текущей строки
\033[s	Сохраняет текущую позицию курсора
\033[u	Восстанавливает сохраненную позицию курсора

Используя эти коды, можно сконструировать строку приглашения, рисующую красный прямоугольник с часами в верхней части экрана (время отображается желтым цветом). Код, определяющий строку приглашения, на этот раз выглядит немного устрашающе:

```
PS1="\[\033[s\033[0;0H\033[0;41m\033[K\033[1;33m\t\033[0m\033[u\]<\u@\h \W>\$ "
```

В табл. 13.5 приведены отдельные части этого определения, они помогут понять, как это работает.

Таблица 13.5. Экранированные последовательности, управляющие позицией курсора

Последовательность	Действие
\[	Начинает последовательность непечатаемых символов. Истинное назначение этой последовательности — позволить bash правильно вычислить длину строки приглашения на экране. Без этого функция редактирования командной строки неправильно позиционировала бы курсор
\033[s	Сохраняет позицию курсора. Это необходимо, чтобы вернуться в местоположение строки приглашения после вывода прямоугольника с часами в верхней части экрана. Будьте внимательны: некоторые эмуляторы терминалов не поддерживают эту последовательность
\033[0;0H	Перемещает курсор в левый верхний угол, в строку 0, позицию 0
\033[0;41m	Устанавливает красный цвет фона
\033[K	Очищает экран от текущей позиции курсора (в левом верхнем углу) до конца строки. Поскольку теперь установлен красный цвет фона, строка окрашивается в красный цвет. Обратите внимание, что последовательность очистки экрана до конца строки не изменяет позицию курсора, который остается в левом верхнем углу
\033[1;33m	Устанавливает желтый цвет текста

Таблица 13.5 (*окончание*)

Последовательность	Действие
`\t`	Выводит текущее время. Хотя это «печатаемый» элемент, он находится в непечатаемом блоке строки приглашения, потому что нам не нужно, чтобы командная оболочка `bash` учитывала размер часов при расчете истинного размера отображаемой строки приглашения
`\033[0m`	Выключает цвет. Сбрасывает настройки цвета для текста и фона
`\033[u`	Восстанавливает позицию курсора, сохраненную ранее
`\]`	Завершает последовательность непечатаемых символов
`<\u@\h \W>\$`	Строка приглашения

Сохранение определения приглашения

Я думаю, что мало у кого возникло желание вводить это монструозное определение каждый раз, поэтому нам нужно где-то сохранить строку приглашения. Сохранить определение можно в файле *.bashrc*. Для этого добавьте следующие две строки в файл:

```
PS1="\[\033[s\033[0;0H\033[0;41m\033[K\033[1;33m\t\033[0m\033[u\]<\u@\h \W>\$ "
export PS1
```

Заключение

Хотите — верьте, хотите — нет, но со строкой приглашения можно творить чудеса, задействовав функции и сценарии, которые мы пока не рассматривали. Все, что было описано выше, — хорошее начало. Не все захотят возиться с изменением приглашения к вводу, оформление по умолчанию тоже выглядит неплохо. Но тем из вас, кому нравится копаться в мелочах, командная оболочка предоставляет возможность творчески провести несколько часов.

Часть III

ТИПИЧНЫЕ ЗАДАЧИ И ОСНОВНЫЕ ИНСТРУМЕНТЫ

14 Управление пакетами

Общаясь с другими членами сообщества Linux, мы услышим массу мнений о том, какой дистрибутив Linux лучше. Часто обзоры дистрибутивов выглядят довольно глупыми, скатываясь к сравнению, например, привлекательности обоев рабочего стола (некоторые отвергают Ubuntu, потому что им не нравится цветовая схема по умолчанию!) и других тривиальных особенностей.

Самой важной отличительной чертой дистрибутива является система управления пакетами и активность сообщества, поддерживающего дистрибутив. Поработав с Linux достаточно долгое время, легко заметить, насколько динамичен программный ландшафт этой системы. Он находится в постоянном движении. Большинство создателей основных дистрибутивов Linux выпускают новые версии каждые шесть месяцев, а множество отдельных программ обновляется каждый день. Чтобы не отставать от этой лавины программного обеспечения, нам нужен хороший инструмент для *управления пакетами*.

Управление пакетами (package management) — это методика установки и управления программным обеспечением в системе. В наши дни большинство может удовлетворить все свои потребности в программном обеспечении, устанавливая *пакеты*, подготовленные создателями соответствующих дистрибутивов Linux. Это отличается от ситуации, возникшей в первые годы развития Linux, когда для установки программ требовалось загружать и компилировать исходный код. Нельзя сказать, что было бы неправильно устанавливать программы из исходных кодов; в действительности наличие доступа к исходному коду является самым большим достоинством в Linux. Это предоставляет возможность исследовать и улучшать систему. Просто работать с заранее скомпилированными пакетами проще и быстрее.

В этой главе мы рассмотрим некоторые инструменты командной строки, используемые для управления пакетами. В то время как все основные дистрибутивы предоставляют мощные и современные программы с графическим интерфейсом

для управления системой, умение работать с программами командной строки по-прежнему востребовано. Они способны выполнять задачи, многие из которых сложно (если вообще возможно) выполнить с использованием их аналогов с графическим интерфейсом.

Системы пакетов

Разные дистрибутивы используют различные системы пакетов, и, как правило, пакеты, подготовленные для одного дистрибутива, несовместимы с другими. В большинстве дистрибутивов используется одна из двух основных технологий упаковки: разработанная создателями дистрибутива Debian с пакетами *.deb* и разработанная создателями дистрибутива Red Hat с пакетами *.rpm*. Существует несколько важных исключений, таких как Gentoo, Slackware и Arch, но в большинстве других дистрибутивов используется одна из двух основных систем, что показано в табл. 14.1.

Таблица 14.1. Основные системы пакетов

Система пакетов	Дистрибутивы (неполный список)
Debian (*.deb*)	Debian, Ubuntu, Linux Mint, Raspbian
Red Hat (*.rpm*)	Fedora, CentOS, Red Hat Enterprise Linux, OpenSUSE

Как действует система пакетов

Способ распространения программ, используемый в индустрии патентованного программного обеспечения, обычно включает покупку установочного носителя, такого как «установочный диск», и последующий запуск мастера установки нового приложения в систему.

Linux действует иначе. Практически все программное обеспечение для системы Linux находится в Интернете. Большая его часть предоставляется создателями дистрибутивов в форме *файлов пакетов*, а остальная часть доступна в исходном коде, который можно установить вручную. Мы еще поговорим об установке программ путем компиляции исходного кода в главе 23.

Файлы пакетов

Основной единицей программного обеспечения в системе пакетов является *файл пакета*. Файл пакета — это сжатая коллекция файлов, составляющих программный пакет. Пакет может состоять из множества программ и файлов с данными,

поддерживающих программы. Помимо файлов для установки, файл пакета включает также метаданные с информацией о пакете, например текстовым описанием пакета и его содержимого. Дополнительно многие пакеты включают сценарии для выполнения настроек до и после установки пакета.

Файлы пакетов создаются людьми, *ответственными за сопровождение пакетов* (package maintainer), часто (но не всегда) являющимися сотрудниками компании-производителя дистрибутива. Ответственный за пакет получает программное обеспечение в исходном коде от поставщика (автора программы), компилирует его и создает метаданные для пакета вместе со всеми необходимыми сценариями установки. Часто ответственный за сопровождение пакета вносит изменения в оригинальный исходный код с целью улучшения интеграции программы с другими компонентами дистрибутива Linux.

Репозитории

Некоторые проекты самостоятельно создают пакеты и дистрибутивы своего программного обеспечения, и все же большинство пакетов в наше время собирается создателями дистрибутивов и заинтересованными третьими сторонами. Готовые пакеты помещаются в центральный репозиторий дистрибутива, где они становятся доступными для пользователей. Репозиторий может содержать тысячи пакетов, специально собранных для дистрибутива.

Для дистрибутива может поддерживаться несколько разных репозиториев с программным обеспечением, находящимся на разных этапах разработки. Например, дистрибутивы обычно имеют *тестовый репозиторий*, содержащий недавно созданные пакеты, которые предназначены для смельчаков, пытающихся отыскать ошибки до того, как пакеты попадут в основной дистрибутив. Нередко дистрибутивы имеют *репозиторий для разработки*, куда помещаются пакеты, продолжающие разрабатываться и предназначенные для включения в ближайший выпуск дистрибутива.

Дистрибутив может также иметь сторонние репозитории. Они необходимы для распространения программного обеспечения, которое по юридическим причинам, связанным с патентами или законами об управлении цифровыми правами (Digital Rights Management, DRM), не может быть включено в дистрибутив. Самым известным случаем является поддержка шифрования DVD, которая считается незаконной в Соединенных Штатах. Сторонние репозитории располагаются в странах, где патенты или законы, ограничивающие распространение программного обеспечения, отсутствуют или действуют иначе. Эти репозитории обычно полностью независимы от поддерживаемого ими дистрибутива, и для их использования нужно знать об их существовании и вручную включать их в конфигурационные файлы с настройками системы управления пакетами.

Зависимости

Программы редко действуют в одиночку; чаще они полагаются на наличие других программных компонентов. Стандартные операции, такие как ввод/вывод, например, выполняются процедурами, которые совместно используются многими программами. Эти процедуры хранятся в так называемых *разделяемых библиотеках* (shared libraries), предоставляющих важные услуги нескольким программам. Если пакету требуется некий общий ресурс, такой как разделяемая библиотека, про него говорят, что он имеет *зависимость*. Современные системы управления пакетами поддерживают некоторые методы *разрешения зависимостей*, — это гарантирует, что после установки пакета в системе будут также установлены все его зависимости.

Высоко- и низкоуровневые инструменты управления пакетами

Системы управления пакетами обычно включают инструменты двух типов:

- низкоуровневые инструменты, решающие такие задачи, как установка и удаление файлов пакетов;
- и высокоуровневые инструменты, выполняющие поиск в метаданных и разрешение зависимостей.

В этой главе мы посмотрим, какие инструменты входят в состав систем на основе Debian (таких, как Ubuntu и многих других), а также в состав последних продуктов Red Hat. Несмотря на то что все дистрибутивы на основе Red Hat опираются на одну и ту же низкоуровневую программу (`rpm`), они используют разные высокоуровневые инструменты. В ходе обсуждения мы познакомимся с высокоуровневой программой `yum`, используемой в дистрибутивах Fedora, Red Hat Enterprise Linux и CentOS. Другие дистрибутивы на основе Red Hat предоставляют высокоуровневые инструменты, сопоставимые по своим возможностям (табл. 14.2).

Таблица 14.2. Инструменты управления пакетами

Дистрибутивы	Низкоуровневые инструменты	Высокоуровневые инструменты
На основе Debian	`dpkg`	`apt-get`, `apt`, `aptitude`
Fedora, Red Hat Enterprise Linux, CentOS	`rpm`	`yum, dnf`

Типичные задачи управления пакетами

С помощью инструментов командной строки для управления пакетами можно выполнить множество разных операций. Мы рассмотрим наиболее типичные из них. Вы должны знать, что низкоуровневые инструменты поддерживают также создание файлов пакетов, но эта тема выходит за рамки данной книги.

В следующем обсуждении под термином `имя_пакета` будет подразумеваться фактическое имя пакета, а под термином `файл_пакета` — имя файла пакета.

Поиск пакета в репозитории

Используя высокоуровневые инструменты для поиска метаданных в репозитории, можно найти пакет по его имени или описанию (табл. 14.3).

Таблица 14.3. Команды поиска пакетов

Дистрибутив	Команды
Debian	`apt-get update` `apt-cache search искомая_строка`
Red Hat	`yum search искомая_строка`

Вот пример поиска текстового редактора `emacs` в системе Red Hat с помощью команды `yum`:

```
yum search emacs
```

Установка пакета из репозитория

Высокоуровневые инструменты позволяют загрузить пакет из репозитория и установить его с полным разрешением всех зависимостей (табл. 14.4).

Таблица 14.4. Команды установки пакетов

Дистрибутив	Команды
Debian	`apt-get update` `apt-cache install имя_пакета`
Red Hat	`yum install имя_пакета`

Вот пример установки текстового редактора `emacs` в системе Debian из репозитория `apt`:

```
apt-get update; apt-get install emacs
```

Установка пакета из файла пакета

Если файл пакета загружен из источника, не являющегося репозиторием, его можно установить непосредственно (без разрешения зависимостей) с использованием низкоуровневого инструмента (табл. 14.5).

Таблица 14.5. Низкоуровневые команды установки пакетов

Дистрибутив	Команды
Debian	`dpkg -i файл_пакета`
Red Hat	`rpm -i файл_пакета`

Пример: если с некоторого сайта, не являющегося репозиторием, был загружен файл *emacs-22.1-7.fc7-i386.rpm*, его можно установить в систему Red Hat командой

```
rpm -i emacs-22.1-7.fc7-i386.rpm
```

ПРИМЕЧАНИЕ

Поскольку этот прием установки основан на использовании низкоуровневой программы `rpm`, он не выполняет разрешения зависимостей. Если программа `rpm` обнаружит неразрешенную зависимость, она завершится с сообщением об ошибке.

Удаление пакета

Пакеты можно удалять с помощью и низкоуровневых, и высокоуровневых инструментов. Примеры использования высокоуровневых инструментов приводятся в табл. 14.6.

Таблица 14.6. Команды удаления пакетов

Дистрибутив	Команды
Debian	`apt-get remove имя_пакета`
Red Hat	`yum erase имя_пакета`

Пример: удалить пакет `emacs` из системы Debian можно командой:

```
apt-get remove emacs
```

Обновление пакетов из репозитория

Наиболее типичной задачей управления пакетами является поддержание системы в актуальном состоянии обновлением пакетов до последних версий. Высокоуровневые инструменты способны выполнять эту важную задачу за один шаг (табл. 14.7).

Таблица 14.7. Команды обновления пакетов

Дистрибутив	Команды
Debian	`apt-get update; apt-get upgrade`
Red Hat	`yum update`

Пример: следующая команда применит все обновления, доступные для пакетов, установленных в системе на основе Debian:

```
apt-get update; apt-get upgrade
```

Обновление пакета из файла пакета

Если обновленная версия пакета была загружена из источника, не являющегося репозиторием, ее можно установить, заменив предыдущую версию (табл. 14.8).

Таблица 14.8. Низкоуровневые команды обновления пакетов

Дистрибутив	Команды
Debian	`dpkg -i файл_пакета`
Red Hat	`rpm -U файл_пакета`

Пример: обновить установленную программу `emacs` до версии, содержащей в файле пакета *emacs-22.1-7.fc7-i386.rpm*, в системе Red Hat можно командой

```
rpm -U emacs-22.1-7.fc7-i386.rpm
```

ПРИМЕЧАНИЕ

`dpkg` не имеет параметра, отвечающего за обновление пакета вместо установки, как в программе `rpm`.

Список установленных пакетов

Команды в табл. 14.9 можно использовать для вывода списка всех пакетов, установленных в системе.

Таблица 14.9. Команды вывода списка пакетов

Дистрибутив	Команды
Debian	`dpkg -l`
Red Hat	`rpm -qa`

Определение, установлен ли пакет

С помощью низкоуровневых инструментов из табл. 14.10 можно определить, был ли установлен определенный пакет.

Таблица 14.10. Команды определения состояния пакетов

Дистрибутив	Команды
Debian	`dpkg -s имя_пакета`
Red Hat	`rpm -q имя_пакета`

Пример: определить, был ли установлен пакет `emacs` в системе Debian, можно командой

```
dpkg -s emacs
```

Вывод информации об установленном пакете

Если известно имя установленного пакета, с помощью команд из табл. 14.11 можно получить описание пакета.

Таблица 14.11. Команды получения информации о пакетах

Дистрибутив	Команды
Debian	`apt-cache show имя_пакета`
Red Hat	`yum info имя_пакета`

Пример: получить описание пакета `emacs` в системе Debian можно командой

```
apt-cache show emacs
```

Поиск пакета по установленному файлу

Определить, в составе какого пакета был установлен некий файл, можно с помощью команд из табл. 14.12.

Таблица 14.12. Команды идентификации принадлежности файлов

Дистрибутив	Команды
Debian	`dpkg -S имя_файла`
Red Hat	`rpm –qf имя_файла`

Пример: узнать, в составе какого пакета был установлен файл `/usr/bin/vim` в системе Red Hat, можно командой

```
rpm -qf /usr/bin/vim
```

Заключение

В последующих главах мы исследуем множество программ, решающих широкий спектр прикладных задач. Хотя большинство этих программ обычно устанавливается по умолчанию, иногда возникает необходимость установить дополнительные пакеты. С вновь обретенными знаниями (и пониманием) особенностей управления пакетами вы без труда сможете установить дополнительные программы и управлять ими.

МИФ ОБ УСТАНОВКЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ В LINUX

Те, кто прежде использовал другие платформы, иногда становятся жертвами мифов о сложности установки программного обеспечения в Linux и верят, что многообразие систем управления пакетами, используемых разными дистрибутивами, является серьезной помехой. Вообще-то и правда — помехой, только не для пользователей, а для производителей патентованного программного обеспечения, желающих распространять свои программы только в виде двоичных файлов.

Программная экосистема Linux основана на идеологии открытости исходного кода. Если разработчик программного обеспечения выпустит исходный код своего продукта, почти наверняка человек, связанный с дистрибутивом, упакует этот продукт и включит его в репозиторий. Этот подход гарантирует хорошую интеграцию продукта с дистрибутивом, и пользователь сможет получить все необходимое ему программное обеспечение в одном месте, вместо того чтобы искать отдельные программы по разным веб-сайтам. В последнее время крупные поставщики проприетарных платформ начали создавать магазины приложений, имитирующие эту идею.

Драйверы устройств распространяются почти так же, только они не выделяются в отдельные пакеты в репозитории дистрибутива, а включаются в ядро Linux. Можно сказать, что в Linux нет такого понятия, как «диск с драйверами». Либо ядро поддерживает данное устройство, либо нет, а ядро Linux поддерживает огромное число устройств. В действительности намного больше, чем Windows. Конечно, едва ли вас утешит информация, что нужное вам устройство не поддерживается ядром. Однако если такое случится, ищите причину. Отсутствие драйвера поддержки обычно обусловлено одной из следующих причин:

- **Устройство слишком новое.** Так как многие производители аппаратного обеспечения не очень активно поддерживают Linux, задача написать драйвер для включения в ядро ложится на членов сообщества Linux. А это требует времени.
- **Устройство чересчур экзотическое.** Не все дистрибутивы включают все возможные драйверы устройств. Для каждого дистрибутива настраивается свое ядро, и так как ядра настраиваются до мелочей (благодаря чему открывается возможность использовать Linux в самых разных устройствах, от наручных часов до больших ЭВМ), создатели дистрибутива могли пропустить ваше устройство. Найдя и загрузив исходный код драйвера, вы (да, да — вы) сможете скомпилировать и установить драйвер самостоятельно. Это не очень сложно, скорее утомительно. О компиляции программного обеспечения мы поговорим в последующих главах.
- **Производители аппаратного обеспечения что-то скрывают.** Производитель не выпустил либо исходный код драйвера для Linux, либо документацию, на основе которой можно было бы написать драйвер. Это означает, что производитель аппаратного обеспечения пытается сохранить программные интерфейсы устройства в секрете. Так как мы предпочитаем не использовать засекреченные устройства в своих компьютерах, я предлагаю удалить это нетолерантное устройство и отправить его в кучу из других бесполезных гаджетов.

15 Устройства хранения

В предыдущих главах мы познакомились с приемами работы с данными на уровне файлов. В этой главе мы будем рассматривать данные на уровне устройств. Linux обладает удивительными возможностями работы с устройствами хранения, такими как жесткие диски, сетевые хранилища или виртуальные устройства хранения, например RAID (Redundant Array of Independent Disks — избыточный массив из независимых дисков) и LVM (Logical Volume Manager — диспетчер логических томов).

Однако поскольку эта книга не о системном администрировании, мы не будем пытаться охватить эту тему во всех подробностях, а всего лишь познакомимся с некоторыми понятиями и ключевыми командами, которые используются для управления устройствами хранения данных.

Для выполнения упражнений к этой главе нам понадобится флеш-диск (флешка), подключаемый к порту USB компьютера и диск CD-RW (для систем, оборудованных пишущим приводом CD-ROM).

Мы познакомимся со следующими командами:

`mount` — монтирует файловые системы;

`umount` — размонтирует файловые системы;

`fsck` — проверяет и восстанавливает файловые системы;

`fdisk` — инструмент для работы с таблицей разделов;

`mkfs` — создает файловые системы;

`dd` — выполняет запись данных блоками непосредственно в устройство;

`genisoimage` (`mkisofs`) — создает файл образа ISO 9660;

`wodim` (`cdrecord`) — записывает данные на оптический носитель;

`md5sum` — вычисляет контрольную сумму MD5.

Монтирование и размонтирование устройств хранения

Последние достижения Linux на настольных компьютерах сделали управление устройствами хранения чрезвычайно простым для обычных пользователей. Достаточно подключить устройство к компьютеру, и оно тут же готово к работе. Раньше (года этак до 2004-го) все необходимые операции требовалось выполнять вручную. В серверных системах эти операции по большей части все еще выполняются вручную, потому что серверы часто предъявляют особые требования к устройствам хранения и настройкам.

Первый шаг в управлении устройствами хранения — подключение самого устройства к дереву файловой системы. Этот процесс называется *монтированием* и позволяет устройству участвовать в работе операционной системы. Как рассказывалось в главе 2, Unix-подобные операционные системы, такие как Linux, поддерживают единое дерево файловой системы, к разным точкам которого подключаются дополнительные устройства. Этот подход отличается от используемого в других операционных системах, таких как Windows, где каждому устройству соответствует отдельное дерево файлов и каталогов (например, `C:\`, `D:\` и т. д.).

В файле с именем */etc/fstab* (сокращенно от «file system table» — таблица файловых систем) перечисляются устройства (обычно разделы жесткого диска), монтируемые на этапе загрузки. Ниже приводится пример содержимого */etc/fstab* из системы Fedora:

```
LABEL=/12        /                      ext4    defaults        1 1
LABEL=/home      /home                  ext4    defaults        1 2
LABEL=/boot      /boot                  ext4    defaults        1 2
tmpfs            /dev/shm               tmpfs   defaults        0 0
devpts           /dev/pts               devpts  gid=5,mode=620  0 0
sysfs            /sys                   sysfs   defaults        0 0
proc             /proc                  proc    defaults        0 0
LABEL=SWAP-sda3  swap                   swap    defaults        0 0
```

Большинство файловых систем из перечисленных в приведенном примере являются виртуальными, и наше обсуждение к ним неприменимо. Наибольший интерес для нас в рамках исследования данной темы представляют первые три:

```
LABEL=/12        /                      ext4    defaults        1 1
LABEL=/home      /home                  ext4    defaults        1 2
LABEL=/boot      /boot                  ext4    defaults        1 2
```

Это разделы жесткого диска. Каждая строка включает шесть полей, описание которых приводится в табл. 15.1.

Таблица 15.1. Поля в файле /etc/fstab

Поле	Содержит	Описание
1	Устройство	Традиционно это поле содержит фактическое имя файла устройства, связанного с физическим устройством, такое как */dev/hda1* (первый раздел ведущего (master) устройства на первом канале IDE). Но, учитывая, что в современных компьютерах может быть множество динамически подключаемых устройств (таких, как устройства USB), многие современные дистрибутивы Linux связывают устройства с текстовыми метками. Такая метка (записываемая в устройство хранения во время форматирования) может быть простым текстом или случайно сгенерированным универсально уникальным идентификатором (Universally Unique Identifier, UUID). Эта метка читается операционной системой в момент подключения устройства. Благодаря этому становится неважно, с каким файлом устройства связано физическое устройство, оно в любом случае будет идентифицировано верно
2	Точка монтирования	Каталог в файловой системе, к которому подключается устройство
3	Тип файловой системы	Linux позволяет монтировать множество типов файловых систем. Наиболее близкой к Linux является файловая система ext4, но точно так же поддерживаются другие типы, такие как FAT16 (`msdos`), FAT32 (`vfat`), NTFS (`ntfs`), CD-ROM (`iso9660`), и пр.
4	Параметры	Файловые системы могут монтироваться с разными параметрами. Например, можно смонтировать файловую систему в режиме «только для чтения» или запретить выполнять какие-либо программы из нее (очень полезная мера предосторожности для съемных носителей)
5	Частота	Единственное число, определяющее, когда должно выполняться резервное копирование файловой системы командой `dump`
6	Порядок	Единственное число, определяющее, в каком порядке файловая система должна проверяться командой `fsck`

Просмотр списка смонтированных файловых систем

Для монтирования файловых систем используется команда `mount`. Если ввести команду без аргументов, она выведет список файловых систем, смонтированных в настоящий момент:

```
[me@linuxbox ~]$ mount
/dev/sda2 on / type ext4 (rw)
```

```
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/sda5 on /home type ext4 (rw)
/dev/sda1 on /boot type ext4 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
fusectl on /sys/fs/fuse/connections type fusectl (rw)
/dev/sdd1 on /media/disk type vfat (rw,nosuid,nodev,noatime, uhelper=hal,uid=500,ut
f8,shortname=lower)
twin4:/musicbox on /misc/musicbox type nfs4 (rw,addr=192.168.1.4)
```

Список имеет следующий формат: `устройство on точка_монтирования type тип_файловой_системы (параметры)`. Например, первая строка соответствует устройству */dev/sda2*, смонтированному как корневая файловая система типа ext4, доступная для чтения и записи (параметр `rw`). В конце списка можно заметить две интересные записи. Предпоследняя запись соответствует 2-гигабайтной SD-карте памяти в устройстве для чтения карт памяти, смонтированной в каталог */media/disk*, последняя запись соответствует сетевому приводу, смонтированному в каталог */misc/musicbox*.

Для первого эксперимента возьмем привод CD-ROM. Сначала посмотрим, что имеется в системе, перед тем как вставить компакт-диск:

```
[me@linuxbox ~]$ mount
/dev/mapper/VolGroup00-LogVol00 on / type ext4 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/hda1 on /boot type ext4 (rw)
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
```

Этот список получен в системе CentOS 5, где для создания корневой файловой системы используется диспетчер LVM. Подобно многим современным дистрибутивам Linux, эта система пытается автоматически монтировать компакт-диски. Вставив в привод компакт-диск, мы увидим следующее:

```
[me@linuxbox ~]$ mount
/dev/mapper/VolGroup00-LogVol00 on / type ext4 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw,gid=5,mode=620)
/dev/hda1 on /boot type ext4 (rw)
```

```
tmpfs on /dev/shm type tmpfs (rw)
none on /proc/sys/fs/binfmt_misc type binfmt_misc (rw)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw)
/dev/sdc on /media/live-1.0.10-8 type iso9660 (ro,noexec,nosuid,nodev,uid=500)
```

Это практически тот же список, с одной дополнительной записью. Последняя запись в списке сообщает, что компакт-диск в приводе CD-ROM (устройство /dev/sdc в этой системе) смонтирован в каталог */media/live-1.0.10-8* и имеет файловую систему `iso9660` (типичную для компакт-дисков). Обратите внимание на имя устройства. Когда вы будете проводить эксперимент в своей системе, очень вероятно, что имя устройства у вас будет отличаться.

ВНИМАНИЕ

В примерах, демонстрируемых ниже, особое внимание обращайте на фактические имена устройств в вашей системе и не используйте имена, приводящиеся в примерах здесь! Также отметьте, что аудиодиск — это не то же самое, что CD-ROM. Аудиодиск не имеет файловой системы и потому не может быть смонтирован в общепринятом смысле.

Теперь, когда мы знаем имя устройства для привода CD-ROM, размонтируем диск и повторно смонтируем его в другой каталог в дереве файловой системы. Для этого необходимо получить права суперпользователя (способом, соответствующим вашей системе) и размонтировать диск командой `umount`:

```
[me@linuxbox ~]$ su -
Password:
[root@linuxbox ~]# umount /dev/sdc
```

Следующий шаг: создать новую *точку монтирования* диска. Точка монтирования — это самый обычный каталог где-то в дереве файловой системы. В таком каталоге нет ничего необычного. Он даже не должен быть пустым каталогом, правда, монтирование устройства в непустой каталог сделает его прежнее содержимое недоступным, пока устройство не будет размонтировано. Итак, создадим новый каталог:

```
[root@linuxbox ~]# mkdir /mnt/cdrom
```

И наконец, смонтируем CD-ROM в новую точку монтирования. Параметр `-t` позволяет указать тип файловой системы:

```
[root@linuxbox ~]# mount -t iso9660 /dev/sdc /mnt/cdrom
```

ПОЧЕМУ ВАЖНО РАЗМОНТИРОВАТЬ УСТРОЙСТВА

Если взглянуть на вывод команды `free`, показывающей статистику использования памяти, можно увидеть статистику с названием *buffers* (буферы). Компьютерные системы проектируются так, чтобы работать максимально быстро. Но медленные устройства препятствуют этому. Ярким примером служат принтеры. Даже самый быстрый принтер выглядит чрезвычайно медлительным по компьютерным стандартам. Компьютеры работали бы крайне медленно, если бы действительно были вынуждены ждать, пока принтер завершит печать страницы. В давние времена (когда персональные компьютеры еще не были многозадачными) это представляло настоящую проблему. При попытке распечатать электронную таблицу или текстовый документ компьютер мог стать недоступным до конца печати. Компьютер не мог посылать данные принтеру быстрее, чем тот мог их обработать, а принтеры не могли работать быстрее, потому что не могли быстро печатать. Эта проблема была решена созданием *буфера печати*, устройства, содержащего некоторый объем ОЗУ и находящегося между компьютером и принтером. При наличии буфера печати компьютер мог послать данные в буфер печати, который сохранял их в быстрой памяти ОЗУ, и компьютер возвращался к работе, не дожидаясь конца печати. В то же время буфер печати мог передавать данные принтеру из своей памяти со скоростью, приемлемой для принтера.

Идея буферизации широко используется для увеличения производительности компьютеров — необходимость работы с медленными устройствами не должна ухудшать производительность системы. Операционные системы хранят данные, прочитанные с устройства и предназначенные для записи в устройство, так долго, насколько это возможно, и используют их, прежде чем фактически обратиться к медленному устройству. В системе Linux, например, можно заметить, что при продолжительной работе она заполняет всю память. Это не означает, что Linux «использует» всю память, это означает лишь то, что Linux использует в своих интересах всю доступную память и буферизует как можно больше данных.

Буферизация позволяет очень быстро выполнять запись в устройства хранения, потому что запись в физическое устройство откладывается «на потом». Данные, предназначенные для устройства, накапливаются в памяти. Время от времени операционная система записывает эти данные в физическое устройство.

Размонтирование устройства влечет за собой запись всех оставшихся данных в это устройство, чтобы его можно было безопасно извлечь. Если носитель извлечь, не выполнив размонтирования, есть вероятность, что не все данные, предназначенные для устройства, будут записаны в него. Иногда эти данные могут включать жизненно важные обновления каталогов, отсутствие которых может привести к *повреждению файловой системы* — одной из самых больших неприятностей, которые могут случиться с компьютером.

После этого можно исследовать содержимое компакт-диска в новой точке монтирования:

```
[root@linuxbox ~]# cd /mnt/cdrom
[root@linuxbox cdrom]# ls
```

Обратите внимание, что происходит при попытке размонтировать компакт-диск:

```
[root@linuxbox cdrom]# umount /dev/sdc
umount: /mnt/cdrom: device is busy
```

В чем причина? Устройство нельзя размонтировать, если оно используется каким-то пользователем или другим процессом. В данном случае мы изменили текущий рабочий каталог, перенеся его в точку монтирования компакт-диска, что и стало причиной занятости устройства. Эту проблему легко исправить, перенеся текущий рабочий каталог куда-нибудь в другое место за пределами точки монтирования:

```
[root@linuxbox cdrom]# cd
[root@linuxbox ~]# umount /dev/hdc
```

Теперь устройство было успешно размонтировано.

Определение названий устройств

Иногда сложно определить название (имя) устройства. В прошлом это было проще. Устройство всегда находилось в одном месте и никогда не менялось. Unix-подобные системы именно так и действовали. Во времена, когда разрабатывалась система Unix, для «смены дискового устройства» требовалось использовать подъемник, чтобы извлечь из комнаты с ЭВМ устройство размером со стиральную машину. В последние годы типовая аппаратная конфигурация настольного компьютера стала намного динамичнее, и система Linux вынуждена быть более гибкой, чем ее предшественницы.

В примерах, приведенных выше, мы использовали способность современной системы Linux «как по волшебству» монтировать устройства, чтобы узнавать их названия постфактум. Но как быть тем, кто управляет сервером или каким-то другим окружением, где автоматическое монтирование не поддерживается? Как в этом случае определить название устройства?

Сначала давайте посмотрим, как система выбирает названия для устройств. Если вывести содержимое каталога */dev* (где живут все устройства), можно увидеть значительное число устройств:

```
[me@linuxbox ~]$ ls /dev
```

Содержимое списка показывает некоторые шаблоны в именовании устройств, неполный список которых приводится в табл. 15.2.

Таблица 15.2. Названия устройств хранения данных в Linux

Шаблон	Устройство
*/dev/fd**	Приводы гибких дисков
*/dev/hd**	Диски IDE (PATA) в старых системах. Обычно материнские платы содержат два разъема, или *канала* IDE, к каждому из которых можно подключить шлейф, рассчитанный на два устройства. Первое устройство, подключенное к такому шлейфу, называется *ведущим устройством* (master device), а второе — *подчиненным устройством* (slave device). Имена устройств упорядочены так, что ведущее устройство, подключенное к первому каналу, получает имя */dev/hda*, ведомое устройство, подключенное к первому каналу, получает имя */dev/hdb*; ведущее устройство, подключенное ко второму каналу, получает имя */dev/hdc*, и т. д. Цифра в конце определяет номер раздела на устройстве. Например, имя */dev/hda1* соответствует первому разделу на первом жестком диске в системе, тогда как имя */dev/hda* соответствует всему устройству в целом
*/dev/lp**	Принтеры
*/dev/sd**	Диски SCSI. В последних версиях системы Linux ядро интерпретирует все дисковые устройства (включая жесткие диски PATA/SATA, флеш-диски и съемные накопители USB, такие как портативные музыкальные плееры и цифровые камеры) как диски SCSI. В остальном система именования напоминает прежнюю систему */dev/hd**, описанную выше
*/dev/sr**	Приводы оптических дисков (приводы CD/DVD, как пишущие, так и нет)

Кроме того, во многих системах можно увидеть такие символические ссылки, как */dev/cdrom*, */dev/dvd* и */dev/floppy*, которые ссылаются на фактические файлы устройств и предусмотрены для удобства.

Если вам доведется работать в системе, которая не монтирует автоматически съемные носители, вы можете использовать следующий прием для определения названий таких устройств после их подключения. Сначала запустите мониторинг содержимого файла */var/log/messages* или */var/log/syslog* в режиме реального времени (для этого могут потребоваться права суперпользователя):

```
[me@linuxbox ~]$ sudo tail -f /var/log/messages
```

Эта команда выведет несколько последних строк из файла и приостановится. Далее подключите извлекаемое устройство. В этом примере мы использовали

16-мегабайтный флеш-диск. Практически сразу же ядро обнаружит новое устройство и проверит его:

```
Jul 23 10:07:53 linuxbox kernel: usb 3-2: new full speed USB device using uhci_hcd
and address 2
Jul 23 10:07:53 linuxbox kernel: usb 3-2: configuration #1 chosen from 1 choice
Jul 23 10:07:53 linuxbox kernel: scsi3 : SCSI emulation for USB Mass Storage
devices
Jul 23 10:07:58 linuxbox kernel: scsi scan: INQUIRY result too short (5), using 36
Jul 23 10:07:58 linuxbox kernel: scsi 3:0:0:0: Direct-Access Easy Disk 1.00 PQ: 0
ANSI: 2
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] 31263 512-byte hardware sectors
(16 MB)
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Write Protect is off
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Assuming drive cache: write
through
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] 31263 512-byte hardware sectors
(16 MB)
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Write Protect is off
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Assuming drive cache: write
through
Jul 23 10:07:59 linuxbox kernel: sdb: sdb1
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Attached SCSI removable disk
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: Attached scsi generic sg3 type 0
```

Когда вывод опять приостановится, нажмите `CTRL+C`, чтобы вернуться в приглашение командной строки. Наибольший интерес для нас представляют строки с упоминанием имени устройства `[sdb]`, соответствующего нашим ожиданиям в отношении названия устройства диска SCSI. Следующие две строки являются для нас особенно показательными:

```
Jul 23 10:07:59 linuxbox kernel: sdb: sdb1
Jul 23 10:07:59 linuxbox kernel: sd 3:0:0:0: [sdb] Attached SCSI removable disk
```

Они сообщают, что имя */dev/sdb* соответствует всему устройству, а имя */dev/sdb1* — первому разделу на этом устройстве. Как видите, работая с Linux, иногда приходится проводить массу интересных детективных расследований!

СОВЕТ

Прием с использованием команды `tail -f /var/log/messages` демонстрирует отличный способ наблюдения за происходящим в системе в режиме реального времени.

Зная имя устройства, можно смонтировать флеш-диск:

```
[me@linuxbox ~]$ sudo mkdir /mnt/flash
[me@linuxbox ~]$ sudo mount /dev/sdb1 /mnt/flash
[me@linuxbox ~]$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda2             15115452   5186944   9775164  35% /
/dev/sda5             59631908  31777376  24776480  57% /home
/dev/sda1               147764     17277    122858  13% /boot
tmpfs                   776808         0    776808   0% /dev/shm
/dev/sdb1                15560         0     15560   0% /mnt/flash
```

Имя устройства сохраняется неизменным, пока оно остается физически подключенным к компьютеру и до перезагрузки компьютера.

Создание новых файловых систем

Представьте, что вам нужно отформатировать флеш-диск и вместо файловой системы FAT32 создать на нем файловую систему, родную для Linux. Для этого следует выполнить две операции:

1. Создать (при необходимости) новое распределение разделов, если имеющееся вас не устраивает.
2. Создать новую, пустую файловую систему.

ВНИМАНИЕ

Следующее упражнение производит форматирование флеш-диска. Используйте диск, не содержащий ничего, что вам было бы нужно, потому что вся информация на диске будет стерта! И снова: убедитесь, что используете имя устройства, верное для вашей системы, а не то, которое показано в примере. Игнорирование этого предупреждения может привести к форматированию (то есть к стиранию) другого диска!

Управление разделами с помощью fdisk

Программа `fdisk` позволяет напрямую выполнять низкоуровневые операции с дисковыми устройствами (такими, как жесткие диски и флеш-диски). С помощью этого инструмента можно изменять, удалять и создавать разделы на устройстве. Чтобы приступить к работе с флеш-диском, его нужно сначала

размонтировать (если прежде он был смонтирован) и затем запустить программу `fdisk`, как показано ниже:

```
[me@linuxbox ~]$ sudo umount /dev/sdb1
[me@linuxbox ~]$ sudo fdisk /dev/sdb
```

Обратите внимание, что здесь нужно указать имя, соответствующее устройству целиком, то есть всему устройству, без номера раздела. После запуска программы вы увидите следующее приглашение:

```
Команда (m для справки):
```

После ввода команды `m` на экране появится меню программы:

```
Команда описание
   a    изменить флаг загрузочного раздела
   b    изменить вложенную BSD-метку диска
   c    переключить флаг совместимости с DOS
   d    удалить раздел
   l    список известных типов разделов
   m    показать это меню
   n    добавить новый раздел
   p    показать таблицу разделов
   q    выйти без сохранения изменений
   s    создать новую пустую таблицу разделов Sun
   t    изменить тип раздела
   u    изменить единицы отображения/ввода
   v    проверить таблицу разделов
   w    сохранить таблицу на диск и выйти
   x    дополнительные функции (только для экспертов)

Команда (m для справки):
```

Первое, что следует сделать, — исследовать список имеющихся разделов. Для этого введите команду `p`, она выведет таблицу разделов на устройстве:

```
Command (m for help): p
Disk /dev/sdb: 16 MB, 16006656 bytes
1 heads, 31 sectors/track, 1008 cylinders
Units = cylinders of 31 * 512 = 15872 bytes

  Device Boot       Start         End      Blocks   Id  System
/dev/sdb1               2        1008       15608+   b  W95 FAT32
```

Обратите внимание, что устройство имеет объем 16 Мбайт и единственный раздел (1), занимающий 1006 цилиндров из 1008 доступных на устройстве. Раздел

идентифицирован как раздел Windows 95 FAT32. Некоторые программы используют этот идентификатор, ограничивая виды операций с диском, но чаще изменение идентификатора не влечет серьезных последствий. Однако ради демонстрации мы изменим его, чтобы показать, что это раздел Linux. Для этого нужно сначала узнать, какой идентификатор обозначает разделы Linux. В листинге, приведенном выше, мы видели, что существующий раздел имеет идентификатор `b` (столбец `Id`). Чтобы увидеть список известных типов разделов, вернитесь к меню программы и обратите внимание на пункт:

```
l    список известных типов разделов
```

Если ввести команду `l`, появится длинный список допустимых типов разделов. Среди них можно увидеть идентификатор `b` типа существующего раздела и идентификатор `83` для Linux. Вернемся обратно к меню программы, где можно увидеть команду изменения идентификатора раздела:

```
t    изменить тип раздела
```

Введите `t` и затем новый идентификатор:

```
Command (m for help): t
Selected partition 1
Hex code (type L to list codes): 83
Changed system type of partition 1 to 83 (Linux)
```

Это все изменения, которые нам нужно было сделать. До этого момента никаких изменений на самом устройстве не было произведено (все изменения пока просто зафиксированы в памяти программы, а не на физическом устройстве), поэтому теперь запишем измененную таблицу разделов на устройство и выйдем. Для этого введите команду `w`:

```
Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.

WARNING: If you have created or modified any DOS 6.x
partitions, please see the fdisk manual page for additional
information.
Syncing disks.
[me@linuxbox ~]$
```

Если бы мы решили оставить устройство в неизменном состоянии, то могли бы ввести команду `q` и покинуть программу без записи изменений на устройство.

Предупреждающее сообщение, выглядящее зловещим, можно просто игнорировать[1].

Создание новой файловой системы с помощью mkfs

Завершив редактирование разделов (довольно простое, хотя так бывает не всегда), мы создадим на флеш-диске новую файловую систему. Для этого воспользуемся программой `mkfs` (сокращенно от *make filesystem* — создать файловую систему), способной создавать разные файловые системы. Чтобы создать на устройстве файловую систему ext4, следует передать команде параметр `-t` с типом файловой системы ext4, затем указать имя устройства и раздел, который требуется отформатировать:

```
[me@linuxbox ~]$ sudo mkfs -t ext4 /dev/sdb1
mke2fs 2.23.2 (12-Jul-2011)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
3904 inodes, 15608 blocks
780 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=15990784
2 block groups
8192 blocks per group, 8192 fragments per group
1952 inodes per group
Superblock backups stored on blocks:
    8193

Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 34 mounts or
180 days, whichever comes first. Use tune2fs -c or -i to override.
[me@linuxbox ~]$
```

Когда выбирается тип файловой системы ext4, программа выводит массу информации. Чтобы восстановить на устройстве оригинальную файловую систему FAT32, следует указать тип файловой системы `vfat`:

```
[me@linuxbox ~]$ sudo mkfs -t vfat /dev/sdb1
```

[1] Оно гласит: «ВНИМАНИЕ: если вы создали или изменили разделы DOS 6.x, прочитайте дополнительную информацию на странице справочного руководства для команды fdisk». — *Примеч. пер.*

Эту процедуру с редактированием разделов и форматированием можно повторять с любыми дополнительными устройствами хранения, подключаемыми к системе. Хотя в данном примере мы работали с маленьким флеш-диском, ту же процедуру можно применить и к внутренним жестким дискам, и к другим извлекаемым устройствам хранения, таким как жесткие USB-диски.

Проверка и восстановление файловой системы

Знакомясь с файлом */etc/fstab*, мы видели некие странные цифры в конце каждой строки. Каждый раз, когда система загружается, она проверяет целостность файловых систем перед их монтированием. Эту проверку выполняет программа `fsck` (сокращенно от *filesystem check* — проверка файловой системы). Последнее число в каждой записи в файле `fstab` определяет порядок проверки файловых систем. В примере, приведенном выше, видно, что корневая файловая система проверяется первой, вслед за ней проверяются файловые системы *home* и *boot*. Устройства с нулем в последнем поле не проверяются стандартными механизмами.

Программа `fsck` может не только проверить целостность, но и восстановить поврежденные файловые системы с той или иной степенью успеха в зависимости от масштаба повреждений. В Unix-подобных системах восстановленные фрагменты файлов помещаются в каталог *lost+found*, находящийся в корне каждой файловой системы.

Проверить наш флеш-диск (который предварительно необходимо размонтировать) можно с помощью следующей команды:

```
[me@linuxbox ~]$ sudo fsck /dev/sdb1
fsck 1.40.8 (13-Mar-2016)
e2fsck 1.40.8 (13-Mar-2016)
/dev/sdb1: clean, 11/3904 files, 1661/15608 blocks
```

В настоящее время файловые системы повреждаются крайне редко, если нет никаких проблем с аппаратной частью, таких как выход из строя привода диска. В большинстве файловых систем обнаруженные на этапе загрузки повреждения вызывают остановку системы с выводом предложения запустить `fsck` перед продолжением.

ЧТО ТАКОЕ FSCK?

В культуре Unix слово «fsck» часто используется взамен распространенного ругательства, в котором три буквы совпадают с буквами в имени команды. Это показательно — вы почти наверняка будете произносить упомянутое слово, оказавшись в ситуации, вынуждающей запустить `fsck`.

Непосредственное перемещение данных между устройствами

Обычно на компьютерах мы работаем с данными, организованными в файлы, однако точно так же можно работать с данными в «низкоуровневой» форме. Если взглянуть на содержимое диска, можно увидеть, что оно состоит из множества «блоков» данных, которые операционная система интерпретирует как файлы и каталоги. Если бы мы умели интерпретировать диски как простые коллекции блоков данных, мы смогли бы выполнять множество полезных задач, таких как клонирование дисков.

Эту задачу решает программа `dd`. Она копирует блоки данных из одного места в другое. По историческим причинам команда имеет уникальный синтаксис:

```
dd if=входной_файл of=выходной_файл [bs=размер_блока [count=число_блоков]]
```

ВНИМАНИЕ

`dd` — очень мощная команда. Ее название происходит от *data definition* (определение данных), но иногда его расшифровывают как *destroy disk* (уничтожить диск), потому что пользователи часто допускают ошибки в параметрах `if` и `of`. Всегда дважды проверяйте их, прежде чем нажать ENTER!

Представьте, что у вас есть два флеш-диска USB одинакового размера и вам нужно создать точную копию первого диска на втором. Допустим, что после подключения к компьютеру им назначаются имена устройств */dev/sdb* и */dev/sdc* соответственно. В этом случае скопировать содержимое первого диска на второй можно следующей командой:

```
dd if=/dev/sdb of=/dev/sdc
```

Как вариант, если к компьютеру подключено только первое устройство, можно скопировать его содержимое в обычный файл, который впоследствии использовать для восстановления или копирования:

```
dd if=/dev/sdb of=flash_drive.img
```

Создание образа компакт-диска

Запись на компакт-диски (CD-R или CD-RW) выполняется в два этапа.

- Создается файл образа ISO, являющийся точным образом файловой системы компакт-диска.
- Файл образа записывается на носитель (то есть на сам компакт-диск).

Создание образа-копии компакт-диска

Чтобы создать ISO-образ имеющегося компакт-диска, необходимо с помощью `dd` прочитать все блоки с данными с этого компакт-диска и скопировать их в локальный файл. Например, допустим, что у нас есть компакт-диск с дистрибутивом Ubuntu, и мы хотим создать файл ISO-образа, который потом можно будет использовать для создания нескольких копий. Вставив компакт-диск в привод CD-ROM и определив имя устройства (пусть это будет */dev/cdrom*), мы сможем создать файл ISO-образа следующим способом:

```
dd if=/dev/cdrom of=ubuntu.iso
```

Этот прием также применим к дискам DVD с данными, но он не будет работать с аудиодисками, так как для хранения данных на них файловая система не используется. Если вы хотите скопировать аудиодиск, обратитесь к команде `cdrdao`.

Создание образа из коллекции файлов

Создать файл ISO-образа, включающий содержимое некоего каталога, можно с помощью программы `genisoimage`. Для этого сначала создадим каталог со всеми необходимыми файлами для включения в образ и затем командой `genisoimage` создадим файл образа. Например, если предположить, что вы создали каталог *~/cd-rom-files* и наполнили его файлами для записи на компакт-диск, следующая команда создаст файл образа с именем *cd-rom.iso*:

```
genisoimage -o cd-rom.iso -R -J ~/cd-rom-files
```

Параметр `-R` требует добавить метаданные *расширений Rock Ridge*, позволяющих использовать длинные имена файлов и права доступа к файлам в стиле POSIX. Аналогично, параметр `-J` включает *расширения Joliet*, разрешающие использовать длинные имена файлов в Windows.

ЧТО В ИМЕНИ ТВОЕМ...

В руководствах по созданию и записи оптических дисков, таких как CD-ROM и DVD, которых в избытке на просторах Интернета, часто можно встретить упоминание двух программ, `mkisofs` и `cdrecord`. Эти программы были некогда частью популярного пакета `cdrtools`, созданного Йоргом Шиллингом (Jörg Schilling). Летом 2006-го мистер Шиллинг изменил лицензию в части, касающейся пакета `cdrtools`, из-за чего она, по мнению многих в сообществе пользователей Linux, стала несовместимой с GNU GPL. Как результат, на основе `cdrtools` был создан альтернативный проект, включающий программы `wodim` и `genisoimage` взамен `cdrecord` и `mkisofs` соответственно.

Запись образа компакт-диска

После подготовки файла образа его можно записать на оптический носитель. Большинство команд, обсуждаемых ниже, применимы и для записи на носители CD-ROM и DVD.

Непосредственное монтирование файла ISO-образа

Существует один трюк, позволяющий монтировать ISO-образы, хранящиеся на жестком диске, и работать с ними, как если бы это были оптические носители. Параметр `-o loop`, добавленный в команду `mount` (вместе с обязательным параметром `-t iso9660`, определяющим тип файловой системы), позволяет смонтировать файл образа в дерево файловой системы, как если бы это было обычное устройство:

```
mkdir /mnt/iso_image
mount -t iso9660 -o loop image.iso /mnt/iso_image
```

В примере, приведенном выше, мы создали точку монтирования с именем */mnt/iso_image* и затем смонтировали в нее файл образа *image.iso*. После монтирования образа с ним можно работать как с настоящим диском CD-ROM или DVD. Не забудьте размонтировать образ, когда он станет не нужен.

Очистка перезаписываемых компакт-дисков

Перезаписываемые компакт-диски CD-RW нужно *стирать*, или *очищать*, перед повторным использованием. Для этого воспользуемся командой `wodim`, указав ей имя устройства пишущего привода компакт-дисков и тип очистки. Программа `wodim` предлагает несколько типов очистки. Для минимальной (и самой быстрой) очистки следует указать тип `fast`:

```
wodim dev=/dev/cdrw blank=fast
```

Запись образа

Записать образ можно с помощью все той же программы `wodim`, указав ей имя устройства пишущего привода компакт-дисков и имя файла образа:

```
wodim dev=/dev/cdrw image.iso
```

Помимо имени устройства и файла образа программа wodim поддерживает массу дополнительных параметров. Чаще других используются параметры -v (обеспечивает вывод подробной информации в ходе записи) и -dao (выполняет запись на диск в режиме *disc-at-once* — диск целиком). Режим «диск целиком» следует использовать, если вы собираетесь воспроизводить диски в коммерческих целях. По умолчанию wodim использует режим *track-at-once* (по одной дорожке), который хорошо подходит для записи музыкальных треков.

Заключительное замечание

В этой главе мы познакомились с основами управления устройствами хранения данных. Эту тему можно было бы продолжать и продолжать. Linux поддерживает широкий спектр устройств хранения и файловых систем, а также предлагает разнообразные возможности для взаимодействий с другими системами.

Дополнительные сведения

Часто бывает полезно проверить целостность ISO-образа, загруженного из Интернета. В большинстве случаев распространители ISO-образов сопровождают их файлами с контрольными суммами. Контрольная сумма — это результат экзотических математических вычислений в виде числа, представляющего содержимое целевого файла. Если содержимое файла образа изменится хотя бы в одном бите, его контрольная сумма будет отличаться от указанной распространителем. Для вычисления контрольной суммы чаще всего используется программа md5sum, возвращающая уникальное шестнадцатеричное число:

```
md5sum image.iso
34e354760f9bb7fbf85c96f6a3f94ece   image.iso
```

Загрузив образ, запустите md5sum для него и сравните результат работы md5sum со значением, указанным распространителем.

Помимо проверки целостности загруженного файла, программу md5sum можно использовать для проверки вновь записанного оптического носителя. Для этого сначала вычислите контрольную сумму для файла образа, а затем — для носителя. Вся хитрость проверки носителя заключается в том, чтобы ограничить вычисления частью оптического носителя, содержащей образ. Для этого определите число

2048-байтных блоков в образе (запись на оптические носители всегда выполняется блоками по 2048 байт) и прочитайте с носителя ровно столько блоков. Для некоторых типов носителей это не обязательно. Например, компакт-диск, записанный в режиме disc-at-once, можно проверить так:

```
md5sum /dev/cdrom
34e354760f9bb7fbf85c96f6a3f94ece    /dev/cdrom
```

Многие типы носителей, такие как DVD, требуют точного вычисления числа блоков. Следующий пример демонстрирует проверку целостности файла образа *dvd-image.iso* и диска в устройстве */dev/dvd* привода DVD. Вам понятно, как работает эта команда?

```
md5sum dvd-image.iso; dd if=/dev/dvd bs=2048 count=$(( $(stat -c "%s" dvd-image.
iso) / 2048 )) | md5sum
```

16 Сети

Когда дело доходит до сетевых возможностей, трудно представить что-то, что было бы невозможно для Linux. Linux используется для создания всех видов сетевых систем, программных компонентов и устройств, включая брандмауэры, маршрутизаторы, серверы имен, сетевые устройства хранения данных (Network-Attached Storage, NAS) и так далее и тому подобное.

Насколько обширна тема сетей, настолько же обширна коллекция команд, которые можно использовать для настройки и управления ими. Мы сосредоточим свое внимание лишь на тех из них, которые чаще всего используются на практике. В число команд, выбранных для исследования в этой главе, входят команды, используемые для мониторинга сетей и передачи файлов. Дополнительно мы исследуем программу `ssh`, используемую для входа в удаленные системы. В этой главе рассматриваются следующие команды:

`ping` — посылает пакеты ICMP ECHO_REQUEST узлам в сети;

`traceroute` — выводит трассировку маршрута пакетов к сетевому узлу;

`ip` — отображает информацию о маршрутах, устройствах, политиках маршрутизации и туннелях и позволяет управлять ими;

`netstat` — выводит список сетевых соединений, таблицы маршрутов, статистику интерфейсов, маскируемые соединения и сведения о членстве в широковещательных группах;

`ftp` — программа передачи файлов через Интернет;

`wget` — неинтерактивный загрузчик файлов из сети;

`ssh` — клиент OpenSSH SSH (программа для входа в удаленные системы).

Далее предполагается, что вы имеете некоторые базовые знания о сетях. В нашем мире повсеместного распространения Интернета каждый пользователь компьютера должен иметь представление о том, как действуют сети, хотя бы на элементарном уровне. Для полноценного использования этой главы вы должны знать следующие термины:

- IP-адрес (адрес протокола Интернета).
- Имя хоста и домена.
- URI (Uniform Resource Identifier — унифицированный идентификатор ресурса).

ПРИМЕЧАНИЕ

Для доступа к некоторым командам из рассматриваемых ниже может потребоваться установить дополнительные пакеты (в зависимости от дистрибутива) из репозиториев вашего дистрибутива, и некоторые из них могут требовать привилегий суперпользователя.

Исследование и мониторинг сети

Даже если вы не являетесь системным администратором, бывает полезно уметь оценивать производительность и функционирование сети.

ping

Команда `ping` является самой простой сетевой командой. Она посылает специальные сетевые пакеты IMCP ECHO_REQUEST указанному сетевому узлу. Большинство сетевых устройств принимает эти пакеты и отвечает на них, — это позволяет проверить сетевые соединения.

ПРИМЕЧАНИЕ

Многие сетевые устройства (в том числе и компьютеры с Linux) могут настраиваться так, чтобы игнорировать эти пакеты. Обычно это делается для повышения безопасности и отчасти — чтобы ввести в заблуждение потенциального злоумышленника. Кроме того, многие брандмауэры блокируют трафик IMCP.

Например, с помощью команды `ping` можно проверить достижимость сетевого узла *http://www.linuxcommand.org/* (один из моих любимых сайтов ;-)):

```
[me@linuxbox ~]$ ping linuxcommand.org
```

Сразу после запуска программа `ping` начинает посылать пакеты с определенным интервалом (по умолчанию 1 секунда), пока ее выполнение не будет прервано:

```
[me@linuxbox ~]$ ping linuxcommand.org
PING linuxcommand.org (66.35.250.210) 56(84) bytes of data.
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=1 ttl=43 time=107 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=2 ttl=43 time=108 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=3 ttl=43 time=106 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=4 ttl=43 time=106 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=5 ttl=43 time=105 ms
64 bytes from vhost.sourceforge.net (66.35.250.210): icmp_seq=6 ttl=43 time=107 ms

--- linuxcommand.org ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 6010ms
rtt min/avg/max/mdev = 105.647/107.052/108.118/0.824 ms
```

После прерывания нажатием CTRL+C (в данном примере после шестого пакета) `ping` выводит результаты своей работы. Если сеть функционирует должным образом, число потерянных пакетов (packet loss) будет составлять ноль процентов. Успешная работа `ping` может служить признаком того, что сетевые компоненты (интерфейсные карты, кабели, маршрутизаторы и шлюзы) находятся в рабочем состоянии.

traceroute

Программа `traceroute` (в некоторых системах используется похожая на нее программа `tracepath`) выводит список всех «переходов» (hops) на пути сетевого трафика между локальной системой и указанным узлом сети. Например, увидеть, как выглядит маршрут к сайту *http://www.slashdot.org/*, можно с помощью следующей команды:

```
[me@linuxbox ~]$ traceroute slashdot.org
```

Ее вывод выглядит примерно так:

```
traceroute to slashdot.org (216.34.181.45), 30 hops max, 40 byte packets
 1 ipcop.localdomain (192.168.1.1) 1.066 ms 1.366 ms 1.720 ms
 2 * * *
```

```
 3 ge-4-13-ur01.rockville.md.bad.comcast.net (68.87.130.9) 14.622 ms 14.885 ms
15.169 ms
 4 po-30-ur02.rockville.md.bad.comcast.net (68.87.129.154) 17.634 ms 17.626 ms
17.899 ms
 5 po-60-ur03.rockville.md.bad.comcast.net (68.87.129.158) 15.992 ms 15.983 ms
16.256 ms
 6 po-30-ar01.howardcounty.md.bad.comcast.net (68.87.136.5) 22.835 ms 14.23 3 ms
14.405 ms
 7 po-10-ar02.whitemarsh.md.bad.comcast.net (68.87.129.34) 16.154 ms 13.600 ms
18.867 ms
 8 te-0-3-0-1-cr01.philadelphia.pa.ibone.comcast.net (68.86.90.77) 21.951 ms 21.073
ms 21.557 ms
 9 pos-0-8-0-0-cr01.newyork.ny.ibone.comcast.net (68.86.85.10) 22.917 ms 21 .884 ms
22.126 ms
10 204.70.144.1 (204.70.144.1) 43.110 ms 21.248 ms 21.264 ms
11 cr1-pos-0-7-3-1.newyork.savvis.net (204.70.195.93) 21.857 ms cr2-pos-0-0-3-1.
newyork.savvis.net (204.70.204.238) 19.556 ms cr1-pos-0-7-3-1.newyork.savvis.net
(204.70.195.93) 19.634 ms
12 cr2-pos-0-7-3-0.chicago.savvis.net (204.70.192.109) 41.586 ms 42.843 ms cr2-
tengig-0-0-2-0.chicago.savvis.net (204.70.196.242) 43.115 ms
13 hr2-tengigabitethernet-12-1.elkgrovech3.savvis.net (204.70.195.122) 44.215 ms
41.833 ms 45.658 ms
14 csr1-ve241.elkgrovech3.savvis.net (216.64.194.42) 46.840 ms 43.372 ms 47.041 ms
15 64.27.160.194 (64.27.160.194) 56.137 ms 55.887 ms 52.810 ms
16 slashdot.org (216.34.181.45) 42.727 ms 42.016 ms 41.437 ms
```

Здесь можно видеть, что на пути между нашей тестовой системой и *http://www.slashdot.org/* находится 16 маршрутизаторов. Для маршрутизаторов, предоставляющих идентификационную информацию, выводятся имена хостов, IP-адреса и информация о производительности, которая включает три интервала времени, понадобившихся для передачи/подтверждения пакетов между локальной системой и маршрутизатором. Для маршрутизаторов, не предоставляющих идентификационной информации (например, из-за особенностей настройки маршрутизатора, заторов в сети, действий брандмауэров и т. д.), выводятся звездочки, как это можно видеть в строке, соответствующей второму переходу. Иногда, когда информация о маршрутизаторе блокируется, это ограничение можно преодолеть, передав команде `traceroute` параметр `-T` или `-I`.

ip

Программа `ip` — это многофункциональный инструмент для настройки параметров подключения к сети, использующий весь спектр сетевых функций, доступных в современных ядрах Linux. Она пришла на замену ныне устаревшей

программе ifconfig. С помощью ip можно исследовать сетевые интерфейсы и таблицу маршрутизации системы.

```
[me@linuxbox ~]$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever

2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
group default qlen 1000
    link/ether ac:22:0b:52:cf:84 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.14/24 brd 192.168.1.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::ae22:bff:fe52:cf84/64 scope link
       valid_lft forever preferred_lft forever
```

Пример, приведенный выше, показывает, что наша тестовая система имеет два сетевых интерфейса. Первый, с именем lo, — это *петлевой интерфейс* (loopback), виртуальный интерфейс, который система использует, чтобы разговаривать «сама с собой»; второй, с именем eth0, — это интерфейс Ethernet.

Выполняя причинно-следственную диагностику, первое, на что следует обратить внимание, — наличие слова UP в первой строке с информацией о каждом интерфейсе, указывающего, что сетевой интерфейс включен, и присутствие допустимого IP-адреса в поле inet во третьей строке. Для систем, использующих протокол динамической настройки хостов (Dynamic Host Configuration Protocol, DHCP), наличие допустимого IP-адреса в этом поле подтвердит нормальную работу DHCP.

netstat

Программа netstat используется для исследования различных настроек сети и статистик. С помощью множества параметров этой команды можно просматривать самые разные аспекты настройки сети. С помощью параметра -ie, например, можно исследовать сетевые интерфейсы в системе:

```
[me@linuxbox ~]$ netstat -ie
eth0    Link encap:Ethernet HWaddr 00:1d:09:9b:99:67
        inet addr:192.168.1.2 Bcast:192.168.1.255 Mask:255.255.255.0
```

```
        inet6 addr: fe80::21d:9ff:fe9b:9967/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
        RX packets:238488 errors:0 dropped:0 overruns:0 frame:0
        TX packets:403217 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:100
        RX bytes:153098921 (146.0 MB) TX bytes:261035246 (248.9 MB)
        Memory:fdfc0000-fdfe0000

lo      Link encap:Local Loopback
        inet addr:127.0.0.1 Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
        UP LOOPBACK RUNNING MTU:16436 Metric:1
        RX packets:2208 errors:0 dropped:0 overruns:0 frame:0
        TX packets:2208 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:111490 (108.8 KB) TX bytes:111490 (108.8 KB)
```

Использование параметра `-r` позволит получить таблицу маршрутизации ядра. По этой таблице можно судить, как настроена передача пакетов между сетями:

```
[me@linuxbox ~]$ netstat -r
Таблица маршутизации ядра протокола IP
Destination  Gateway     Genmask       Flags  MSS Window  irtt Iface
192.168.1.0  *           255.255.255.0 U        0      0     0 eth0
default      192.168.1.1 0.0.0.0       UG       0      0     0 eth0
```

В этом простом примере представлена типичная таблица маршрутизации для клиентской машины, подключенной к локальной сети (Local Area Network, LAN), находящейся за брандмауэром/маршрутизатором. В первой строке демонстрируется адрес назначения `192.168.1.0`. IP-адреса, оканчивающиеся нулем, соответствуют целым сетям, а не отдельным узлам в них, поэтому такой адрес подразумевает: «любой узел в локальной сети». Следующее поле, `Gateway` (шлюз), определяет имя или IP-адрес шлюза (маршрутизатора) для выхода текущего узла в указанную сеть. Звездочка в этом поле указывает, что использовать шлюз не требуется.

В последней строке в качестве адреса назначения указано слово `default` (по умолчанию). Эта строка управляет трафиком, адресованным любым сетям, не перечисленным в таблице. В данном примере видно, что роль шлюза выполняет маршрутизатор с адресом `192.168.1.1`, который, по всей видимости, знает, что делать с трафиком.

Так же как `ip`, программа `netstat` имеет множество параметров, из которых мы рассмотрели только пару. Полный их список вы найдете на странице справочного руководства (man) для `netstat`.

Передача файлов по сети

Что толку от сети, если не знать, как перемещать файлы через нее? Существует множество программ, перемещающих данные по сети. Сейчас мы рассмотрим две из них, а еще несколько — в последующих разделах.

ftp

Одна из по-настоящему «классических» программ — `ftp` — получила свое имя от используемого ею протокола, *протокола передачи файлов* (File Transfer Protocol, FTP). Протокол FTP широко используется в Интернете для передачи файлов. Он поддерживается большинством веб-браузеров, если не всеми, и вам часто будут встречаться идентификаторы URI, начинающиеся с префикса протокола `ftp://`.

Программа `ftp` появилась задолго до веб-браузеров. Она использовалась для обмена данными с серверами FTP, компьютерами, хранящими файлы, которые можно выгружать и загружать по сети.

Протокол FTP (в своем первоначальном виде) небезопасен, потому что пересылает имена и пароли в открытом текстовом виде. То есть они не шифруются, и любой, кто способен перехватить сетевой трафик, сможет увидеть их. По этой причине практически все операции по протоколу FTP в Интернете выполняются анонимными северами FTP. Анонимный сервер позволяет любому желающему подключиться с учетной записью *anonymous* без пароля.

В следующем примере показан типичный сеанс работы с программой `ftp` для загрузки ISO-образа с дистрибутивом Ubuntu из каталога */pub/cd_images/Ubuntu-18.04* анонимного сервера FTP *fileserver*.

```
[me@linuxbox ~]$ ftp fileserver
Connected to fileserver.localdomain.
220 (vsFTPd 2.0.1)
Name (fileserver:me): anonymous
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd pub/cd_images/Ubuntu-18.04
250 Directory successfully changed.
ftp> ls
200 PORT command successful. Consider using PASV.
```

```
150 Here comes the directory listing.
-rw-rw-r--    1 500      500      733079552 Apr 25 03:53 ubuntu-18.04-desktop-
amd64.iso
226 Directory send OK.
ftp> lcd Desktop
Local directory now /home/me/Desktop
ftp> get ubuntu-18.04-desktop-amd64.iso
local: ubuntu-18.04-desktop-amd64.iso remote: ubuntu-18.04-desktop-amd64.iso
200 PORT command successful. Consider using PASV.
150 Opening BINARY mode data connection for ubuntu-18.04-desktop-amd64.iso
(733079552 bytes).
226 File send OK.
733079552 bytes received in 68.56 secs (10441.5 kB/s)
ftp> bye
```

В табл. 16.1 приводится описание команд, вводившихся в ходе этого сеанса.

Таблица 16.1. Примеры команд интерактивного сеанса ftp

Команда	Значение
`ftp fileserver`	Вызывает программу `ftp` и предлагает ей подключиться к серверу *fileserver*
`anonymous`	Имя пользователя для входа. После ввода имени пользователя появится приглашение ввести пароль. Некоторые серверы принимают пустой пароль. Другие могут требовать пароль в формате адреса электронной почты. В данном случае попробуйте ввести что-нибудь похожее на *user@example.com*
`cd pub/cd_images/Ubuntu-18.04`	Выполняет переход в каталог в удаленной системе, где находится требуемый файл. Обратите внимание, что на большинстве анонимных серверов FTP файлы, доступные для загрузки любым желающим, находятся где-то в каталоге *pub*
`ls`	Выводит содержимое каталога в удаленной системе
`lcd Desktop`	Выполняет переход в каталог *~/Desktop* в локальной системе. В этом примере программа `ftp` была вызвана в текущем рабочем каталоге ~. Данная команда назначает текущим рабочий каталог *~/Desktop*

Команда	Значение
get ubuntu-18.04-desktop-amd64.iso	Посылает удаленной системе запрос на передачу файла *ubuntu-18.04-desktop-amd64.iso* локальной системе. Так как в локальной системе текущим рабочим выбран каталог *~/Desktop*, файл будет загружен в него
bye	Выходит из удаленной системы и завершает сеанс программы ftp. Так же можно использовать команды quit и exit

Если в приглашении ftp> ввести команду help, программа выведет список поддерживаемых команд. С помощью программы ftp можно выполнять множество обычных операций с файлами на сервере, правда, при наличии достаточных привилегий. Это не очень удобно, но выполнимо.

lftp — более удачная версия ftp

ftp — не единственный клиент FTP командной строки. В действительности таких клиентов множество. Одним из лучших (и более популярным) считается lftp Александра Лукьянова (Alexander Lukyanov). Этот клиент действует почти так же, как традиционная программа ftp, но имеет множество дополнительных функций, включая поддержку нескольких протоколов (в том числе и HTTP), возможность автоматического восстановления прервавшейся загрузки, выполнение операций в фоновом режиме, автодополнение путей по клавише Tab и многое другое.

wget

wget — еще одна популярная программа командной строки для загрузки файлов. Ее удобно использовать для загрузки содержимого веб- и FTP-сайтов. С помощью wget можно загрузить один файл, несколько файлов и даже целый сайт. Например, загрузить первую страницу сайта *http://www.linuxcommand.org/* можно командой:

```
[me@linuxbox ~]$ wget http://linuxcommand.org/index.php
--11:02:51-- http://linuxcommand.org/index.php
           => `index.php'
Распознаётся linuxcommand.org... 66.35.250.210
Подключение к linuxcommand.org|66.35.250.210|:80... соединение установлено.
```

```
HTTP-запрос отправлен. Ожидание ответа… 200 OK
Длина: 3808 (3.7K) [text/html]
Сохранение в: "index.php"

    [ <=>                                ] 3,120 --.--K/s

11:02:51 (161.75 MB/s) - `index.php' сохранён [3808/30808]
```

Большинство параметров, поддерживаемых программой `wget`, позволяет организовать рекурсивную загрузку, загрузку файлов в фоновом режиме (позволяет выйти из системы без остановки загрузки) и догружать частично загруженные файлы. Эти возможности хорошо описаны на странице справочного руководства (man).

Безопасные взаимодействия с удаленными узлами

Уже много лет Unix-подобные операционные системы поддерживают возможность удаленного администрирования по сети. На первом этапе, еще до повсеместного распространения Интернета, существовала пара популярных программ для входа в удаленные сетевые узлы: `rlogin` и `telnet`. Однако эти программы страдали тем же фатальным недостатком, что и программа `ftp`; все данные (включая имена пользователей и пароли) они передавали в виде открытого текста. Это совершенно недопустимо в эпоху Интернета.

ssh

Для решения описанной проблемы был разработан протокол с названием SSH (Secure Shell — безопасная командная оболочка). SSH решает две основные проблемы безопасного взаимодействия с удаленными сетевыми узлами:

- подтверждает, что удаленный узел является именно тем, за кого себя выдает (это предотвращает атаки вида «злоумышленник в середине» (man-in-the-middle));
- шифрует все данные, передаваемые между локальным и удаленным узлами.

В своей работе протокол SSH опирается на два компонента. На удаленном узле действует сервер SSH, принимающий соединения на порте 22, а в локальной системе действует клиент SSH, осуществляющий обмен информацией с удаленным сервером.

Большинство дистрибутивов Linux включают реализацию SSH с названием `OpenSSH` из проекта BSD. Некоторые дистрибутивы (например, Red Hat) по умолчанию содержат пакеты с обоими компонентами, сервером и клиентом, тогда как другие (например, Ubuntu) включают только клиента. Чтобы система могла принимать удаленные соединения, в ней должен быть установлен пакет с реализацией сервера `OpenSSH-server`, этот сервер должен быть настроен и запущен, и если система находится за брандмауэром, последний должен пропускать входящие соединения на порт TCP с номером 22.

СОВЕТ

Если у вас нет удаленной системы, с которой можно было бы устанавливать соединения, но вы желаете поработать с примерами, приведенными ниже, установите пакет `OpenSSH-server` в своей системе и используйте имя `localhost` в качестве имени удаленного узла. В этом случае ваш компьютер будет устанавливать соединения с самим собой.

Программа клиента SSH, используемая для подключения к серверам SSH, имеет достаточно очевидное имя: `ssh`. Подключиться к удаленному сетевому узлу с именем `remote-sys` можно с помощью программы клиента `ssh`, как показано ниже:

```
[me@linuxbox ~]$ ssh remote-sys
The authenticity of host 'remote-sys (192.168.1.4)' can't be established.
RSA key fingerprint is 41:ed:7a:df:23:19:bf:3c:a5:17:bc:61:b3:7f:d9:bb.
Are you sure you want to continue connecting (yes/no)?[1]
```

При первой попытке подключения на экран выводится предупреждение, сообщающее, что аутентичность удаленного узла не может быть установлена. Это объясняется тем, что программа-клиент прежде никогда не подключалась к данному удаленному узлу. Чтобы принять идентификационные данные удаленного узла, введите `yes` в ответ на приглашение. После установки соединения пользователю будет предложено ввести пароль:

```
Warning: Permanently added 'remote-sys,192.168.1.4' (RSA) to the list of known
hosts.
me@remote-sys's password:[2]
```

[1] Аутентичность узла ‘remote-sys (192.168.1.4)’ не может быть установлена. Идентификационный ключ RSA: 41:ed:7a:df:23:19:bf:3c:a5:17:bc:61:b3:7f:d9:bb. Вы уверены, что желаете установить соединение (да/нет)? — *Примеч. пер.*

[2] Внимание: узел ‘remote-sys,192.168.1.4’ (RSA) добавлен в хранимый список известных узлов. — *Примеч. пер.*

После ввода действительного пароля в терминале появится приглашение командной оболочки из удаленной системы:

```
Last login: Tue Aug 30 13:00:48 2018
[me@remote-sys ~]$
```

Сеанс с удаленной командной оболочкой продолжается, пока пользователь не введет команду `exit` в приглашении удаленной командной оболочки, после чего соединение закроется. В этот момент возобновится сеанс локальной командной оболочки и появится ее приглашение к вводу.

К удаленной системе можно также подключиться с другим именем пользователя. Например, если локальный пользователь `me` имеет в удаленной системе учетную запись с именем `bob`, он сможет войти в удаленную систему с именем `bob`, выполнив следующую команду:

```
[me@linuxbox ~]$ ssh bob@remote-sys
bob@remote-sys's password:
Last login: Tue Aug 30 13:03:21 2018
[bob@remote-sys ~]$
```

Как отмечалось выше, `ssh` проверяет аутентичность удаленного узла. Если удаленный узел не пройдет аутентификацию, появится следующее предупреждение:

```
[me@linuxbox ~]$ ssh remote-sys
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
41:ed:7a:df:23:19:bf:3c:a5:17:bc:61:b3:7f:d9:bb.
Please contact your system administrator.
Add correct host key in /home/me/.ssh/known_hosts to get rid of this message.
Offending key in /home/me/.ssh/known_hosts:1
RSA host key for remote-sys has changed and you have requested strict
checking.
Host key verification failed.[1]
```

[1] Перевод:

```
ВНИМАНИЕ: ИЗМЕНИЛСЯ ИДЕНТИФИКАТОР УДАЛЕННОГО УЗЛА!
ЕСТЬ ВЕРОЯТНОСТЬ, ЧТО КТО-ТО ЗАМЫСЛИЛ ЧТО-ТО НЕДОБРОЕ!
Кто-то может подслушивать вас прямо сейчас (атака "злоумышленник в середине")!
Возможно также, что просто изменился идентификационный ключ RSA узла.
```

Это сообщение появляется в двух возможных ситуациях. Первая: злоумышленник мог попытаться провести атаку вида «злоумышленник в середине». Это случается редко, потому что все знают, что `ssh` предупреждает пользователя об этом. Более вероятная причина связана с некими изменениями в удаленной системе: например, была выполнена переустановка операционной системы или сервера SSH. Однако в интересах безопасности не следует сбрасывать со счетов первую возможность. Всегда обращайтесь к системному администратору удаленной системы, когда появится это сообщение.

Убедившись в безобидности причин, вызвавших это сообщение, можно исправить проблему на стороне клиента. Для этого с помощью текстового редактора (например, `vim`) удалите устаревший ключ из файла *~/.ssh/known_hosts*. В примере сообщения выше присутствует строчка:

```
Offending key in /home/me/.ssh/known_hosts:1[1]
```

Это означает, что подозреваемый ключ хранится в строке 1, в файле *known_hosts*. Удалите эту строку из файла и позвольте программе `ssh` принять новые идентификационные данные от удаленной системы.

Помимо открытия сеанса командной оболочки в удаленной системе `ssh` позволяет также выполнить единственную команду. Например, в удаленной системе `remote-sys` можно выполнить команду `free` и получить результаты в локальной системе:

```
[me@linuxbox ~]$ ssh remote-sys free
me@twin4's password:
             total     used     free   shared   buffers    cached
Mem:        775536   507184   268352        0    110068    154596
-/+ buffers/cache:   242520   533016
Swap:      1572856        0  1572856
[me@linuxbox ~]$
```

Этот прием открывает возможность для довольно интересных вариантов использования, как в следующем примере, где вывод команды `ls` в удаленной системе перенаправляется в локальный файл:

```
Удаленный узел прислал идентификационный ключ RSA:
41:ed:7a:df:23:19:bf:3c:a5:17:bc:61:b3:7f:d9:bb.
Пожалуйста, свяжитесь со своим системным администратором.
Добавьте правильный ключ узла в /home/me/.ssh/known_hosts, чтобы избавиться от этого
сообщения.
Подозреваемый ключ хранится в файле /home/me/.ssh/known_hosts:1
Идентификационный ключ RSA узла remote-sys изменился, а вы запросили строгую проверку.
Проверка ключа удаленного узла завершилась неудачей.
```
— *Примеч. пер.*

[1] Подозреваемый ключ хранится в файле /home/me/.ssh/known_hosts:1. — *Примеч. пер.*

```
[me@linuxbox ~]$ ssh remote-sys 'ls *' > dirlist.txt
me@twin4's password:
[me@linuxbox ~]$
```

Обратите внимание на одиночные кавычки. Они необходимы для предотвращения подстановки пути в локальной системе; нам требуется, чтобы подстановка была выполнена в удаленной системе. Аналогично, если бы нам потребовалось перенаправить вывод в файл в удаленной системе, мы могли бы поместить оператор перенаправления и имя файла внутрь одиночных кавычек:

```
[me@linuxbox ~]$ ssh remote-sys 'ls * > dirlist.txt'
```

СОЗДАНИЕ ТУННЕЛЯ SSH

При установке SSH-соединения с удаленным узлом между локальной и удаленной системами создается *шифрованный туннель*. Обычно этот туннель используется для безопасной передачи команд из локальной системы в удаленную и безопасной передачи результатов обратно. Помимо этой основной задачи, протокол SSH позволяет также передавать через шифрованный туннель самые разные виды сетевого трафика, создавая своего рода *виртуальную частную сеть* (Virtual Private Network, VPN) между локальной и удаленной системами.

Чаще всего, пожалуй, эта возможность используется для передачи трафика X Window System. Из системы с действующим X-сервером (то есть отображающей графический интерфейс) можно запустить программу-клиента X (приложение с графическим интерфейсом) в удаленной системе и отображать ее интерфейс в локальной системе. Как это делается, показано в следующем примере. Представьте, что мы работаем в системе Linux с именем `linuxbox`, где запущен X-сервер, и нам понадобилось запустить программу `xload` в удаленной системе с именем `remote-sys` так, чтобы графический интерфейс программы отображался в локальной системе. Добиться этого можно следующим способом:

```
[me@linuxbox ~]$ ssh -X remote-sys
me@remote-sys's password:
Last login: Mon Sep 05 13:23:11 2018
[me@remote-sys ~]$ xload
```

После запуска программы `xload` в удаленной системе ее окно появится в локальной системе. В некоторых системах может понадобиться использовать параметр `-Y` вместо `-X`.

scp и sftp

Пакет OpenSSH включает еще две программы, способные использовать шифрованный туннель SSH для копирования файлов по сети. Первая, `scp` (secure copy — безопасное копирование), используется для копирования файлов, как уже знакомая вам программа `cp`. Основное отличие заключается в необходимости предварять пути к исходному или конечному файлу именем удаленного узла и символом двоеточие за ним. Например, скопировать документ с именем *document.txt* из домашнего каталога в удаленной системе `remote-sys` в текущий рабочий каталог в локальной системе можно так:

```
[me@linuxbox ~]$ scp remote-sys:document.txt .
me@remote-sys's password:
document.txt                            100% 5581    5.5KB/s    00:00
[me@linuxbox ~]$
```

По аналогии с командой `ssh` перед именем удаленного узла можно указать имя пользователя, если имя учетной записи в удаленной системе не совпадает с именем учетной записи в локальной системе:

```
[me@linuxbox ~]$ scp bob@remote-sys:document.txt .
```

Вторая программа копирования файлов через SSH-соединение — `sftp`. Как следует из ее имени — это безопасная замена для программы `ftp`. `sftp` действует практически так же, как оригинальная программа `ftp`, которую мы использовали выше, только передает данные не в открытом текстовом виде, а через шифрованный туннель SSH. `sftp` имеет важное преимущество перед обычной программой `ftp` — она не требует, чтобы на удаленном узле работал сервер FTP. Ей необходим только сервер SSH. Это означает, что любой компьютер, к которому можно подключиться с помощью клиента SSH, можно также использовать в качестве FTP-подобного сервера. Ниже приводится пример сеанса работы с программой `sftp`:

```
[me@linuxbox ~]$ sftp remote-sys
Connecting to remote-sys...
me@remote-sys's password:
sftp> ls
ubuntu-8.04-desktop-i386.iso
sftp> lcd Desktop
sftp> get ubuntu-8.04-desktop-i386.iso
Fetching /home/me/ubuntu-8.04-desktop-i386.iso to ubuntu-8.04-desktop-i386.iso
/home/me/ubuntu-8.04-desktop-i386.iso 100% 699MB    7.4MB/s    01:35
sftp> bye
```

СОВЕТ

Протокол SFTP поддерживается многими диспетчерами файлов с графическим интерфейсом, входящими в состав дистрибутивов Linux. В GNOME или KDE можно ввести в адресную строку идентификатор URI, начинающийся с `sftp://`, и работать с файлами, хранящимися в удаленной системе с действующим сервером SSH.

СУЩЕСТВУЮТ ЛИ КЛИЕНТЫ SSH ДЛЯ WINDOWS?

Допустим, что вы работаете за компьютером с операционной системой Windows и вам нужно зайти на свой сервер с Linux, чтобы выполнить некую работу. Как быть? Нужно просто установить в Windows программу клиента SSH! Существует довольно много таких программ. Наиболее популярной, пожалуй, является программа PuTTY Симона Тэтхэма (Simon Tatham) и его команды. Программа PuTTY отображает окно терминала и позволяет пользователю Windows открыть сеанс SSH (или telnet) с удаленным узлом. Программа также предоставляет аналоги программ `scp` и `sftp`.

Программа PuTTY доступна по адресу *http://www.chiark.greenend.org.uk/~sgtatham/putty/*.

Заключение

В этой главе мы рассмотрели несколько сетевых инструментов, присутствующих в большинстве систем Linux. Учитывая, что Linux широко используется в серверах и сетевых устройствах, существует множество других возможностей, которые можно получить, установив дополнительные программы. Но даже имеющийся базовый набор инструментов позволяет решать массу полезных задач, связанных с работой в сети.

17 Поиск файлов

Блуждая по системе Linux, мы совершенно ясно увидели, что типичная Linux-система содержит множество файлов. В связи с этим возникает вопрос: как искать нужные файлы? Мы уже знаем, что файловая система в Linux организована в соответствии с определенными соглашениями, которые переходили из одного поколения Unix-подобных систем в другое, но огромное число файлов может порождать труднопреодолимую проблему.

В этой главе мы рассмотрим два инструмента для поиска файлов в системе:

`locate` — выполняет поиск файлов по именам;

`find` — выполняет поиск файлов в иерархии каталогов.

Мы также познакомимся с командой, часто используемой вместе с командами поиска файлов для обработки списков с результатами:

`xargs` — конструирует команды на основе данных, полученных из стандартного ввода, и выполняет их.

Дополнительно в этой главе будет представлена пара команд, которые помогут нам в наших исследованиях:

`touch` — изменяет времена, ассоциированные с файлом;

`stat` — выводит статус файла или файловой системы.

locate — простой способ поиска файлов

Программа `locate` выполняет быстрый поиск в базе данных имен файлов и выводит все имена, соответствующие искомой строке. Допустим, к примеру, что нужно

найти все программы с именами, начинающимися с *zip*. Поскольку требуется найти программы, можно предположить, что имя каталога с программами оканчивается на *bin/*. Соответственно можно попробовать выполнить поиск с помощью `locate`, как показано ниже:

```
[me@linuxbox ~]$ locate bin/zip
```

`locate` выполнит поиск в базе данных имен файлов и выведет все имена, содержащие строку *bin/zip*:

```
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

Если к результатам поиска предъявляются более строгие требования, команду `locate` можно объединить с другими инструментами, такими как `grep`, позволяющими осуществить более сложный поиск:

```
[me@linuxbox ~]$ locate zip | grep bin
/bin/bunzip2
/bin/bzip2
/bin/bzip2recover
/bin/gunzip
/bin/gzip
/usr/bin/funzip
/usr/bin/gpg-zip
/usr/bin/preunzip
/usr/bin/prezip
/usr/bin/prezip-bin
/usr/bin/unzip
/usr/bin/unzipsfx
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

Программа `locate` существует уже много лет, и за эти годы было создано несколько ее вариантов, получивших широкое распространение. Два из них, наиболее часто используемые в современных дистрибутивах Linux, — это `slocate` и `mlocate`, которые, впрочем, являются символическими ссылками, указывающими на `locate`.

Разные версии locate имеют пересекающиеся множества параметров. Некоторые поддерживают поиск с использованием регулярных выражений (о которых рассказывается в главе 19) и групповые символы. Загляните на страницу справочного руководства (man) для locate, чтобы определить, какая версия установлена у вас.

> **ОТКУДА БЕРЕТСЯ БАЗА ДАННЫХ ДЛЯ LOCATE?**
>
> В некоторых дистрибутивах при попытке запустить locate сразу после установки она потерпит неудачу, но если попытаться использовать ее на следующий день, все, как ни странно, будет работать как надо. Так в чем же проблема? База данных для locate создается другой программой с именем updatedb. Обычно она периодически запускается как *задание cron*; то есть она запускается системным планировщиком cron через регулярные интервалы времени. В большинстве систем, в состав которых входит locate, программа updatedb запускается один раз в сутки. Поскольку база данных не обновляется непрерывно, скорее всего, locate не находит самые свежие файлы. Чтобы решить эту проблему, запустите программу updatedb вручную от имени суперпользователя.

find — сложный способ поиска файлов

В отличие от программы locate, выполняющей поиск файлов по именам, программа find ищет файлы согласно заданным атрибутам в указанном каталоге (и во вложенных подкаталогах). Далее мы потратим время на исследование команды find, потому что она имеет множество интересных особенностей, с которыми мы неоднократно столкнемся, когда начнем знакомиться с приемами программирования в последующих главах.

В простейшем случае программе find можно передать одно или несколько имен каталогов для поиска. Например, с ее помощью можно получить список содержимого домашнего каталога:

```
[me@linuxbox ~]$ find ~
```

Для большинства активных пользователей она выдаст длинный список. Так как список выводится в стандартный вывод, его можно передать по конвейеру другим программам. Воспользуемся программой wc, чтобы подсчитать число файлов:

```
[me@linuxbox ~]$ find ~ | wc -l
47068
```

Ух ты, вот это мы поработали! Вся прелесть команды find в том, что ее можно использовать для поиска файлов, соответствующих определенным критериям. Она делает это посредством (что немного странно) применения *проверок*, *операций* и *параметров*. Сначала рассмотрим проверки.

Проверки

Допустим, мы хотим получить список каталогов. Для этого добавим в команду следующую проверку:

```
[me@linuxbox ~]$ find ~ -type d | wc -l
1695
```

Добавив проверку `-type d`, мы ограничились поиском только каталогов. Но точно так же можно ограничить поиск только обычными файлами:

```
[me@linuxbox ~]$ find ~ -type f | wc -l
38737
```

В табл. 17.1 перечислены проверки типов файлов, наиболее часто используемые с командой find.

Таблица 17.1. Проверки типов файлов в find

Тип файлов	Описание
b	Специальный файл блочного устройства
c	Специальный файл символьного устройства
d	Каталог
f	Обычный файл
l	Символическая ссылка

Добавив дополнительные проверки, можно выполнять поиск файлов по размеру и имени. Давайте найдем все обычные файлы с именами, соответствующими шаблону `*.JPG`, и имеющие размер больше 1 мегабайта:

```
[me@linuxbox ~]$ find ~ -type f -name "*.JPG" -size +1M | wc -l
840
```

В этом примере мы добавили проверку `-name` с шаблоном имени файла. Обратите внимание, что шаблон заключен в кавычки, чтобы предотвратить подстановку

имен файлов командной оболочкой. Далее мы добавили проверку `-size` со строкой `+1M`. Начальный символ «плюс» указывает, что требуется искать файлы, размер которых превышает указанное число. Начальный символ «минус» изменил бы значение строки на противоположное: «меньше указанного числа». Число без знака означает: «в точности соответствует значению». Буква `M` в конце определяет единицы измерения — мегабайты (Megabytes). В табл. 17.2 перечислены символы, которые можно использовать для обозначения единиц измерения.

Таблица 17.2. Единицы измерения, поддерживаемые командой find

Символ	Единица измерения
`b`	Блоки размером по 512 байт (используется по умолчанию, если иное не указано явно)
`c`	Байты
`w`	2-байтные слова
`k`	Килобайты (Kilobytes, блоки по 1024 байт)
`M`	Мегабайты (Megabytes, блоки по 1 048 576 байт)
`G`	Гигабайты (Gigabytes, блоки по 1 073 741 824 байт)

Команда find поддерживает множество разнообразных проверок. В табл. 17.3 приводится краткое описание наиболее часто используемых из них. Обратите внимание, что в случаях, когда требуется числовой аргумент, допустимо использование формы записи с символами `+` и `-`, обсуждавшейся выше.

Таблица 17.3. Проверки, поддерживаемые командой find

Проверка	Описание
`-cmin n`	Соответствует файлам или каталогам, содержимое или атрибуты которых последний раз изменялись точно `n` минут назад. Чтобы выразить условие «менее `n` минут назад», используйте `-n`; чтобы выразить условие «более `n` минут назад», используйте `+n`
`-cnewer имя`	Соответствует файлам или каталогам, содержимое или атрибуты которых последний раз изменялись позже, чем у файла с указанным именем
`-ctime n`	Соответствует файлам или каталогам, содержимое или атрибуты (то есть разрешения) которых последний раз изменялись более чем `n*24` часов назад
`-empty`	Соответствует пустым файлам и каталогам

Таблица 17.3 (*окончание*)

Проверка	Описание
`-group группа`	Соответствует файлам или каталогам, принадлежащим указанной группе. Группа может задаваться именем или числовым идентификатором группы
`-iname шаблон`	Действует так же, как проверка `-name`, но различает регистр символов
`-inum n`	Соответствует файлам с номером индексного узла (inode) `n`. Эту проверку удобно использовать для поиска всех жестких ссылок на определенный индексный узел
`-mmin n`	Соответствует файлам или каталогам, содержимое которых последний раз изменялось `n` минут назад
`-mtime n`	Соответствует файлам или каталогам, содержимое которых последний раз изменялось `n*24` часов назад
`-name шаблон`	Соответствует файлам и каталогам, имена которых совпадают с указанным шаблоном
`-newer имя`	Соответствует файлам и каталогам, содержимое которых последний раз изменялось позже, чем у файла с указанным именем. Эта проверка может пригодиться в сценариях, выполняющих резервное копирование файлов. Каждый раз, в процессе создания резервной копии, можно обновлять файл (например, файл журнала) и затем с помощью `find` определять, какие файлы изменились с момента последнего обновления
`-nouser`	Соответствует файлам и каталогам, не принадлежащим какому-либо допустимому пользователю. Эту проверку можно использовать для поиска файлов, принадлежащих удаленным учетным записям, или для обнаружения следов злоумышленников
`-nogroup`	Соответствует файлам и каталогам, не принадлежащим какой-либо допустимой группе
`-perm режим`	Соответствует файлам или каталогам с указанным режимом доступа. Режим может задаваться восьмеричным числом или иметь символическую форму
`-samefile имя`	Действует так же, как проверка `-inum`. Соответствует файлам с тем же номером индексного узла (inode), что и файл с указанным именем
`-size n`	Соответствует файлам с размером `n`
`-type c`	Соответствует файлам с типом `c`
`-user имя`	Соответствует файлам или каталогам, принадлежащим пользователю с указанным `именем`. Аргумент `имя` может быть именем или числовым идентификатором пользователя

Это не полный список. Дополнительные детали ищите на странице справочного руководства (man) для команды find.

Операторы

Несмотря на большое число проверок, поддерживаемых командой find, мы все еще нуждаемся в способе определения логических отношений между проверками. Например, представьте, что в некотором каталоге мы хотим найти все файлы и подкаталоги с небезопасными разрешениями. Для этого можно было бы выполнить поиск всех файлов с разрешениями, отличающимися от 0600, и каталогов с разрешениями, отличающимися от 0700. К счастью, find поддерживает возможность комбинирования проверок с помощью логических операторов с целью определить более сложные критерии отбора. Выразить вышеупомянутую проверку можно так:

```
[me@linuxbox ~]$ find ~ \( -type f -not -perm 0600 \) -or \( -type d -not
-perm 0700 \)
```

Ф-фу! Как неизящно! Что все это значит? На самом деле операторы перестанут казаться избыточно сложными, как только вы с ними познакомитесь поближе (табл. 17.4).

Таблица 17.4. Логические операторы, поддерживаемые командой find

Оператор	Описание
-and	Соответствует, если выполняются условия в проверках с обеих сторон от оператора. Можно сократить до -a. Обратите внимание, что в отсутствие операторов по умолчанию подразумевается -and
-or	Соответствует, если выполняется условие с одной из сторон от оператора. Можно сократить до -o
-not	Соответствует, если условие в проверке, следующей за оператором, не выполняется. Можно сократить до -!
( )	Группируют проверки и операторы для формирования крупных выражений. Используются для управления порядком проверок. По умолчанию проверки выполняются слева направо. Часто используются для изменения порядка проверок по умолчанию, чтобы получить желаемый результат. Даже если скобки не нужны, иногда полезно включать их, чтобы сделать команды более наглядными. Не забывайте, что круглые скобки имеют специальное значение для командной оболочки, поэтому их нужно экранировать, чтобы они были переданы команде find как аргументы. Обычно экранирование выполняют с помощью символа «обратный слеш»

Имея список операторов под рукой, попробуем разобрать команду `find`. На самом верхнем уровне мы видим, что проверки объединены в две группы, разделенные оператором `-or`:

```
(выражение 1) -or (выражение 2)
```

В этом есть определенный смысл, потому что мы хотим найти файлы с одним набором разрешений и каталоги — с другим. Но если выполняется поиск *и* файлов, *и* каталогов, почему используется оператор `-or` вместо `-and`? Потому, что `find`, выполняя обход файлов и каталогов, оценивает их по одному, чтобы понять, соответствует ли файл или каталог указанным проверкам. Команде требуется узнать, является ли очередной элемент файлом *или* каталогом с «плохими» разрешениями. Один и тот же элемент не может соответствовать сразу двум условиям. То есть если развернуть сгруппированные выражения, можно увидеть следующее:

```
(файл с плохими разрешениями) -or (каталог с плохими разрешениями)
```

Наша следующая задача — проверить «плохие разрешения». Как это сделать? Фактически никак. Но мы можем проверить «неудовлетворительные разрешения», зная, что такое «удовлетворительные разрешения». В случае с файлами удовлетворительными являются разрешения 0600, для каталогов — 0700. Выражение, проверяющее «неудовлетворительные» разрешения, выглядит так:

```
-type f -and -not -perms 0600
```

а для каталогов так:

```
-type d -and -not -perms 0700
```

Как отмечалось в табл. 17.4, оператор `-and` можно просто удалить, так как он подразумевается по умолчанию. Теперь, объединив все вместе, мы получим окончательную команду:

```
find ~ (-type f -not -perms 0600) -or (-type d -not -perms 0700)
```

Однако поскольку круглые скобки имеют специальное значение для командной оболочки, их нужно экранировать, чтобы предотвратить интерпретацию скобок командной оболочкой. Для этого достаточно добавить обратный слеш перед каждой из них.

Логические операторы имеют еще одну важную особенность, с которой необходимо разобраться. Представьте, что у нас есть два выражения, разделенных логическим оператором:

```
выражение1 -оператор выражение2
```

`Выражение1` будет вычислено в любом случае, а вот будет ли вычислено `выражение2`, зависит от оператора. В табл. 17.5 показано, как это работает.

Таблица 17.5. Действие логических операторов -and/-or команды find

Результат *выражения1*	**Оператор**	***Выражение2***
Истина	`-and`	Всегда вычисляется
Ложь	`-and`	Никогда не вычисляется
Истина	`-or`	Никогда не вычисляется
Ложь	`-or`	Всегда вычисляется

Почему так происходит? Это сделано для повышения производительности. Возьмем для примера оператор `-and`. Мы знаем, что выражение `выражение1 -and выражение2` не может быть истинным, если `выражение1` вернет ложный результат, поэтому нет смысла вычислять `выражение2`. Аналогично, если имеется выражение `выражение1 -or выражение2` и `выражение1` вернет истинный результат, нет смысла вычислять `выражение2`, так как уже известно, что `выражение1 -or выражение2` является истинным.

Это удобно, поскольку такой порядок вычислений помогает повысить скорость выполнения. Но почему это так важно для нас? Потому, что мы можем использовать данную особенность для управления выполнением операций, о которых рассказывается далее.

Предопределенные операции

Давайте попробуем выполнить определенные действия в процессе поиска! Иметь список с результатами работы команды `find` уже неплохо, но представьте, что нам нужно выполнить некие операции с элементами списка. К счастью, `find` позволяет выполнять наши операции, основываясь на результатах поиска. Существует множество предопределенных операций и несколько способов применения операций, определяемых пользователем. Для начала взгляните на неполный список предопределенных операций в табл. 17.6.

Таблица 17.6. Предопределенные операции, поддерживаемые командой find

Операция	**Описание**
`-delete`	Удаляет текущий найденный файл
`-ls`	Действует эквивалентно команде `ls -dils` в отношении найденного файла. Результат выводится в стандартный вывод

Таблица 17.6 (*окончание*)

Операция	Описание
`-print`	Выводит полный путь к найденному файлу в стандартный вывод. Эта операция выполняется по умолчанию, если не указана никакая другая
`-quit`	Завершает выполнение команды после обнаружения первого совпадения

Поддерживаемых операций намного больше, чем показано здесь. Полный список можно найти на странице справочного руководства (man) для команды `find`.

В нашем первом примере мы выполнили команду:

```
find ~
```

Она выводит список всех файлов и подкаталогов, хранящихся в домашнем каталоге. Список выводится просто потому, что в отсутствие других операций предполагается операция `-print`. То есть эту команду можно было бы выразить так:

```
find ~ -print
```

Программу `find` можно использовать для удаления файлов, соответствующих определенным критериям. Например, следующая команда удалит все файлы с расширением *.bak* (которое часто используется для обозначений резервных копий файлов):

```
find ~ -type f -name '*.bak' -delete
```

Эта команда найдет в домашнем каталоге (и во вложенных подкаталогах) пользователя все файлы с расширением *.bak* и удалит их.

ВНИМАНИЕ

Операцию `-delete` следует использовать с особыми предосторожностями. Всегда предварительно проверяйте команду, подставив операцию `-print` вместо `-delete`, чтобы убедиться, что она не удалит ничего лишнего.

Прежде чем продолжить, давайте посмотрим, как логические операторы воздействуют на операции. Взгляните на следующую команду:

```
find ~ -type f -name '*.bak' -print
```

Как видите, эта команда ищет обычные файлы (`-type f`) с расширением *.bak* (`-name '*.bak'`) и выводит относительные пути к ним в стандартный вывод (`-print`). Однако такой порядок работы команды определяется логическими отношениями между всеми проверками и операциями. Как вы помните, между проверками и операциями по умолчанию подразумевается отношение `-and`. Ту же команду можно выразить, добавив логические операторы:

```
find ~ -type f -and -name '*.bak' -and -print
```

Теперь, имея перед глазами это определение, взгляните на табл. 17.7, где показано, как логические операторы влияют на порядок выполнения.

Таблица 17.7. Влияние логических операторов

Проверка/операция	Выполняется, когда
`-print`	`-type f and -name '*.BAK'` истинно
`-name '*.bak'`	`-type f` истинно
`-type f`	Всегда выполняется, потому что это первая проверка/операция в отношении `-and`

Так как логические отношения между проверками и операциями определяют необходимость их выполнения, можно сделать вывод, что их порядок следования играет важную роль. Например, если изменить порядок выполнения операций и проверок, поставив операцию `-print` на первое место, команда будет вести себя иначе:

```
find ~ -print -and -type f -and -name '*.bak'
```

Эта версия команды выведет каждый файл (операция `-print` всегда возвращает истинное значение), а затем проверит тип файла и его расширение.

Операции, определяемые пользователем

Помимо предопределенных операций можно также вызывать произвольные команды. Традиционно с этой целью используется операция `-exec`, что показано ниже:

```
-exec команда {} ;
```

где *команда* — это имя команды, `{}` — символическое представление текущего пути к файлу и точка с запятой — обязательный разделитель, обозначающий конец

команды. Следующий пример демонстрирует использование `-exec` для получения эффекта, аналогичного операции `-delete`, обсуждавшейся выше:

```
-exec rm '{}' ';'
```

И снова, поскольку фигурные скобки и точка с запятой имеют специальное значение для командной оболочки, они должны заключаться в кавычки или экранироваться.

Кроме того, существует возможность выполнять пользовательские операции интерактивно. Если заменить операцию `-exec` операцией `-ok`, перед выполнением каждой указанной команды будет выводиться запрос:

```
find ~ -type f -name 'foo*' -ok ls -l '{}' ';'
< ls ... /home/me/bin/foo > ? y
-rwxr-xr-x 1 me me 224 2007-10-29 18:44 /home/me/bin/foo
< ls ... /home/me/foo.txt > ? y
-rw-r--r-- 1 me me 0 2016-09-19 12:53 /home/me/foo.txt
```

Эта команда ищет файлы с именами, начинающимися со строки `foo`, и для каждого найденного файла выполняет команду `ls -l`. Операция `-ok` запрашивает подтверждение у пользователя, прежде чем выполнить команду `ls`.

Увеличение эффективности

Каждый раз, когда обнаруживается файл, соответствующий критериям, операция `-exec` запускает новый экземпляр указанной команды. Но иногда желательно объединить все результаты поиска и запустить единственный экземпляр команды. Например, вместо последовательности команд, такой как

```
ls -l файл1
ls -l файл2
```

предпочтительнее было бы выполнить команду

```
ls -l файл1 файл2
```

Здесь команда выполняется только один раз, а не несколько. Существует два способа добиться этого: традиционный, с использованием внешней команды `xargs`, и альтернативный, с использованием новой возможности в самой команде `find`. Обсудим сначала альтернативный способ.

Если заменить завершающий символ «точка с запятой» знаком «плюс», в команде find активируется функция объединения результатов в список аргументов для вызова единственного экземпляра требуемой команды. Вернемся к нашему примеру. Команда:

```
find ~ -type f -name 'foo*' -exec ls -l '{}' ';'
-rwxr-xr-x 1 me   me 224 2007-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me   me   0 2016-09-19 12:53 /home/me/foo.txt
```

будет вызывать ls для каждого найденного файла. Изменив команду, как показано ниже:

```
find ~ -type f -name 'foo*' -exec ls -l '{}' +
-rwxr-xr-x 1 me   me 224 2007-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me   me   0 2016-09-19 12:53 /home/me/foo.txt
```

мы получим тот же результат, но система выполнит команду ls только один раз.

xargs

Команда xargs предлагает очень интересную возможность. Она принимает входные данные со стандартного ввода и преобразует их в список аргументов для указанной команды. В данном примере ее можно было бы использовать так:

```
find ~ -type f -name 'foo*' -print | xargs ls -l
-rwxr-xr-x 1 me   me 224 2007-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me   me   0 2016-09-19 12:53 /home/me/foo.txt
```

Здесь вывод команды find передается по конвейеру команде xargs, которая, в свою очередь, конструирует список аргументов для команды ls и выполняет ее.

ПРИМЕЧАНИЕ

Несмотря на то что в командную строку можно включить большое число аргументов, оно не бесконечно. Не исключено, что в результате получится команда, слишком длинная для командной оболочки. Когда длина командной строки превышает максимально допустимый размер, xargs выполнит указанную команду с максимально возможным числом аргументов и затем повторит процесс, пока не исчерпает все, что получит со стандартного ввода. Чтобы увидеть максимально возможную длину командной строки, выполните xargs с параметром --show-limits.

> **ОБРАБОТКА ФАЙЛОВ С НЕОБЫЧНЫМИ ИМЕНАМИ**
>
> Unix-подобные системы позволяют встраивать в имена файлов пробелы (и даже символы перевода строки). Это порождает проблемы при выполнении программ, таких как `xargs`, конструирующих списки аргументов для других программ. Внутренние пробелы интерпретируются как разделители, и получившаяся команда будет интерпретировать слова, разделенные пробелами, как отдельные аргументы. Для решения этой проблемы `find` и `xarg` предлагают использовать в качестве разделителя аргументов *пустой символ* (null character). В кодировке ASCII пустой символ определен как символ с нулевым кодом (в противоположность пробелу, например, который в кодировке ASCII определен как символ с кодом 32). Команда `find` поддерживает операцию `-print0`, которая производит вывод имен файлов, разделенных пустым символом, а команда `xargs` имеет параметр `--null`, позволяющий организовать прием значений, разделенных пустым символом. Например:
>
> ```
> find ~ -iname '*.jpg' -print0 | xargs --null ls -l
> ```
>
> Этот прием гарантирует правильную обработку любых имен файлов, даже содержащих пробелы.

Возвращаемся в песочницу

Пришло время применить `find` для решения некоторых практических (почти) задач. Сначала создадим песочницу с множеством файлов и каталогов:

```
[me@linuxbox ~]$ mkdir -p playground/dir-{001..100}
[me@linuxbox ~]$ touch playground/dir-{001..100}/file-{A..Z}
```

Какая мощь командной строки! Эти две строки создают каталог *playground*, содержащий 100 подкаталогов и 26 пустых файлов в каждом. Попробуйте-ка то же самое сделать в графическом интерфейсе!

Это волшебство мы сотворили с помощью уже знакомой команды (`mkdir`), механизма подстановки в командной оболочке (фигурные скобки) и новой команды `touch`. Объединив команду `mkdir` с параметром `-p` (который вынуждает `mkdir` создать родительские каталоги в указанном пути) с подстановкой фигурных скобок, мы смогли создать 100 каталогов.

Команда `touch` обычно используется для обновления времени последнего изменения файлов. Но если передать ей имя несуществующего файла, она создаст пустой файл.

В нашей песочнице мы создали 100 файлов с именем *file-A*. Давайте найдем их:

```
[me@linuxbox ~]$ find playground -type f -name 'file-A'
```

Обратите внимание, что, в отличие от `ls`, find возвращает результаты в несортированном порядке. Порядок определяется организацией устройства хранения. Мы можем убедиться, что действительно имеем 100 файлов с именем *file-A*:

```
[me@linuxbox ~]$ find playground -type f -name 'file-A' | wc -l
100
```

А теперь выполним поиск файлов по времени их последнего изменения. Этот подход можно использовать для создания резервных копий или организации файлов в хронологическом порядке. Для этого сначала создадим эталонный файл, время последнего изменения которого будет использоваться для сравнения:

```
[me@linuxbox ~]$ touch playground/timestamp
```

Эта команда создаст пустой файл *timestamp* и установит время его последнего изменения равным текущему времени. Мы можем убедиться в этом, использовав еще одну полезную команду, `stat`, которую можно рассматривать как своего рода форсированную версию `ls`. Команда `stat` выводит всю информацию о файле и его атрибутах, которой обладает система:

```
[me@linuxbox ~]$ stat playground/timestamp
  File: `playground/timestamp'
  Size: 0          Blocks: 0       IO Block: 4096 regular empty file
Device: 803h/2051d Inode: 14265061 Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1001/ me)   Gid: ( 1001/ me)
Access: 2018-10-08 15:15:39.000000000 -0400
Modify: 2018-10-08 15:15:39.000000000 -0400
Change: 2018-10-08 15:15:39.000000000 -0400
```

Если применить команду `touch` к файлу еще раз и затем исследовать его с помощью `stat`, мы увидим, что время последнего его изменения обновилось:

```
[me@linuxbox ~]$ touch playground/timestamp
[me@linuxbox ~]$ stat playground/timestamp
  File: `playground/timestamp'
  Size: 0          Blocks: 0       IO Block: 4096 regular empty file
Device: 803h/2051d Inode: 14265061 Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1001/ me)   Gid: ( 1001/ me)
Access: 2018-10-08 15:23:33.000000000 -0400
Modify: 2018-10-08 15:23:33.000000000 -0400
Change: 2018-10-08 15:23:33.000000000 -0400
```

Далее воспользуемся командой `find`, чтобы обновить время последнего изменения некоторых файлов в нашей песочнице:

```
[me@linuxbox ~]$ find playground -type f -name 'file-B' -exec touch '{}' ';'
```

Эта команда обновит время последнего изменения для всех файлов с именем *file-B*, имеющихся в песочнице. Теперь найдем с помощью `find` обновленные файлы, сравнив все файлы с эталонным файлом *timestamp*:

```
[me@linuxbox ~]$ find playground -type f -newer playground/timestamp
```

В результате мы получим все 100 файлов с именем *file-B*. Поскольку команда `touch` применялась ко всем файлам *file-B* в песочнице после обновления файла *timestamp*, они оказались «новее», чем *timestamp*, и потому были идентифицированы проверкой `-newer`.

В заключение вернемся к проверке плохих разрешений, выполнявшейся выше, и применим ее к каталогу *playground*:

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 \) -or \( -type d -not
-perm 0700 \)
```

Эта команда выведет все 100 каталогов и 2600 файлов, хранящихся в *playground* (а также файл *timestamp* и сам каталог *playground*, всего 2702 элемента), потому что ни один из них не соответствует нашему определению «удовлетворительные разрешения». Вооружившись новыми знаниями об операторах и операциях, добавим в эту команду операции для применения новых разрешений к файлам и каталогам в песочнице:

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 -exec chmod 0600
'{}' ';' \) -or \( -type d -not -perm 0700 -exec chmod 0700 '{}' ';' \)
```

Основываясь на повседневном опыте, следует отметить, что намного проще ввести две команды — одну для каталогов и одну для файлов, чем одну большую составную команду, но знание, что можно действовать именно так, вам не помешает. Главное, что вы должны понять, — как можно использовать операторы и операции для решения практических задач.

Параметры

Наконец, мы добрались до параметров. Параметры помогают управлять областью поиска. Они могут включаться в выражения команды `find` наряду с другими проверками и операциями. В табл. 17.8 перечислены наиболее часто используемые параметры.

Таблица 17.8. Параметры команды find

Параметр	Описание
-depth	Требует от find обработать сначала файлы в каталогах и только потом каталоги. Этот параметр автоматически применяется с операцией -delete
-maxdepth число_уровней	Устанавливает максимальное число уровней, на которое команда find может опускаться в дереве каталогов, выполняя проверки и операции
-mindepth число_уровней	Устанавливает минимальное число уровней, на которое команда find должна опуститься в дереве каталогов перед выполнением проверок и операций
-mount	Требует от find не выполнять обход каталогов, в которые смонтированы другие файловые системы
-noleaf	Требует от find не оптимизировать поиск, опираясь на предположение, что поиск ведется в Unix-подобной файловой системе. Этот параметр необходимо использовать при обходе файловых систем DOS/Windows CD-ROM

Заключение

Вы видели сами, как легко определять местоположение файлов (locate) и как сложно искать их (find). Каждая из этих команд имеет свою область применения. Найдите время, чтобы исследовать богатые возможности find. При регулярном использовании эта команда поможет вам лучше понять операции с файловой системой Linux.

18 Архивация и резервное копирование

Одной из основных задач администратора компьютерных систем является обеспечение безопасности данных, а одним из способов решения этой задачи — своевременное создание резервных копий системных файлов. Даже если вы не являетесь системным администратором, вам все равно пригодится умение создавать копии и перемещать большие коллекции файлов из одного места в другое и с одного устройства на другое.

В этой главе мы рассмотрим несколько программ, часто используемых для управления коллекциями файлов:

Программы сжатия:

`gzip` — сжимает и распаковывает файлы;

`bzip2` — программа поблочного сжатия файлов.

Программы архивирования:

`tar` — утилита архивирования на ленту;

`zip` — упаковывает и сжимает файлы.

И программа синхронизации файлов:

`rsync` — выполняет синхронизацию файлов и каталогов с удаленной системой.

Сжатие файлов

На протяжении всей истории развития вычислительных технологий не прекращались попытки размещения большего числа данных в меньшем объеме, будь то

память, устройства хранения или полоса пропускания сети. Многие устройства и технологии, прочно вошедшие в обиход, такие как переносные плееры, телевидение высокой четкости или широкополосный доступ в Интернет, обязаны своим существованием эффективным технологиям сжатия данных.

Сжатие данных — это процесс устранения *избыточных данных*. Давайте рассмотрим воображаемый пример. Допустим, у нас есть файл, хранящий изображение абсолютно черного квадрата размером 100 на 100 пикселов. В терминах хранения данных (если предположить, что каждый пиксел представлен 24 битами, или 3 байтами) изображение занимает 30 000 байт:

$$100 \times 100 \times 3 = 30\,000.$$

Изображение, состоящее из пикселов одного цвета, содержит массу избыточных данных. Будь мы умнее, мы могли бы закодировать данные в виде простого описания того факта, что изображение представлено блоком из 30 000 пикселов черного цвета. То есть вместо хранения блока данных с 30 000 нулей (черный цвет в файлах изображений обычно представлен нулевым значением) мы могли бы сжать данные до числа 30 000 с последующим нулем, описывающим цвет. Такая схема сжатия, она называется *кодированием длин серий* (run-length encoding), является одной из простейших технологий сжатия. Современные технологии не в пример сложнее и эффективнее, но главная цель осталась прежней — избавиться от избыточных данных. *Алгоритмы сжатия* (математические методики, применяемые для осуществления сжатия) делятся на две основные категории:

- *Сжатие без потерь* (lossless) и с потерями (lossy). Сжатие без потерь гарантирует сохранность всех данных, содержащихся в оригинале. То есть после восстановления файла из сжатой версии восстановленный файл будет иметь в точности то же содержимое, что и несжатый оригинал.
- *Сжатие с потерями* (lossy), с другой стороны, удаляет некоторые данные во время сжатия, чтобы обеспечить более высокую степень сжатия. Восстановленный файл в этом случае не будет совпадать с оригинальной версией, скорее он будет близкой аппроксимацией оригинала. Примерами сжатия с потерями могут служить формат JPEG (для изображений) и MP3 (для музыкальных произведений).

В дальнейшем обсуждении мы будем рассматривать только сжатие без потерь, поскольку большинство данных в компьютерах потерь не допускает.

gzip

Программа `gzip` используется для сжатия одного или нескольких файлов. Во время работы она замещает оригинальный файл его сжатой версией. Соответствующая

программа `gunzip` используется для восстановления сжатых файлов до исходного состояния. Например:

```
[me@linuxbox ~]$ ls -l /etc > foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me    me    15738 2018-10-14 07:15 foo.txt
[me@linuxbox ~]$ gzip foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me    me     3230 2018-10-14 07:15 foo.txt.gz
[me@linuxbox ~]$ gunzip foo.txt
[me@linuxbox ~]$ ls -l foo.*
-rw-r--r-- 1 me    me    15738 2018-10-14 07:15 foo.txt
```

В этом примере мы создали текстовый файл с именем *foo.txt*, записав в него список содержимого каталога */etc*. Далее мы запустили программу `gzip`, которая заменила оригинальный файл сжатой версией с именем *foo.txt.gz*. В списке содержимого каталога, который был получен с использованием шаблона *foo.**, можно видеть, что исходный файл действительно был замещен сжатой версией, и эта сжатая версия получилась почти в пять раз меньше оригинала. Можно также заметить, что сжатый файл имеет такие же разрешения и время, что и оригинал.

Далее мы запустили программу `gunzip`, чтобы распаковать файл. После этого, как видите, сжатая версия была замещена оригиналом, и снова с теми же разрешениями и временем.

Программа `gzip` имеет множество параметров, часть которых описана в табл. 18.1.

Таблица 18.1. Параметры команды gzip

Параметр	Длинный параметр	Описание
-c	--stdout --to-stdout	Выводит результат на стандартный вывод и сохраняет оригинальные файлы
-d	--decompress --uncompress	Распаковывает файл. С этим параметром `gzip` действует как `gunzip`
-f	--force	Принудительное (force) сжатие, даже если сжатая версия оригинального файла уже существует
-h	--help	Выводит информацию о порядке использования
-l	--list	Выводит список статистик для каждого сжатого файла
-r	--recursive	Если один или несколько аргументов команды являются каталогами, выполняет рекурсивное сжатие файлов, находящихся в них
-t	--test	Проверяет целостность сжатого файла

Пара-метр	Длинный параметр	Описание
-v	--verbose	Выводит в процессе работы сообщения с информацией о ходе сжатия
-число		Устанавливает степень сжатия. Числом может быть любое целочисленное значение в диапазоне от 1 (высокая скорость работы, низкая степень сжатия) до 9 (низкая скорость работы, высокая степень сжатия). Значения 1 и 9 можно так же заменить параметрами `--fast` и `--best` соответственно. По умолчанию используется значение 6

Вернемся к нашему примеру:

```
[me@linuxbox ~]$ gzip foo.txt
[me@linuxbox ~]$ gzip -tv foo.txt.gz
foo.txt.gz:  OK
[me@linuxbox ~]$ gzip -d foo.txt.gz
```

Здесь мы заменили файл *foo.txt* его сжатой версией с именем *foo.txt.gz*. Затем проверили целостность сжатой версии, передав параметры `-t` и `-v`. В заключение мы распаковали файл, вернув его исходное состояние.

`gzip` можно также использовать несколько необычным способом, через стандартные ввод и вывод:

```
[me@linuxbox ~]$ ls -l /etc | gzip > foo.txt.gz
```

Эта команда создает сжатую версию списка с содержимым каталога.

Программа `gunzip`, которая распаковывает файлы, сжатые с помощью `gzip`, предполагает, что имена файлов оканчиваются расширением *.gz*, поэтому его можно не указывать при условии, что имя файла в команде не соответствует существующему несжатому файлу:

```
[me@linuxbox ~]$ gunzip foo.txt
```

Если цель только в том, чтобы просмотреть содержимое сжатого текстового файла, сделать это можно так:

```
[me@linuxbox ~]$ gunzip -c foo.txt | less
```

ПРИМЕЧАНИЕ

Существует также программа `zless`. Она заменяет собой конвейер, представленный выше.

bzip2

Программа bzip2 Джулиана Сюарда похожа на программу gzip, но использует иной алгоритм, который обеспечивает более высокую степень сжатия ценой снижения скорости работы. Во многих отношениях она действует точно так же, как gzip. Файл, сжатый с помощью bzip2, получает расширение *.bz2*:

```
[me@linuxbox ~]$ ls -l /etc > foo.txt
[me@linuxbox ~]$ ls -l foo.txt
-rw-r--r-- 1 me    me    15738 2018-10-17 13:51 foo.txt
[me@linuxbox ~]$ bzip2 foo.txt
[me@linuxbox ~]$ ls -l foo.txt.bz2
-rw-r--r-- 1 me    me     2792 2018-10-17 13:51 foo.txt.bz2
[me@linuxbox ~]$ bunzip2 foo.txt.bz2
```

Как видите, bzip2 можно использовать так же, как gzip. Все параметры программы gzip (кроме -r), представленные выше, поддерживаются также программой bzip2. Но имейте в виду, что параметр степени сжатия (-число) имеет несколько иной смысл для bzip2. В паре с bzip2 поставляются программы bunzip2 и bzcat для распаковывания файлов.

Существует также программа bzip2recover для восстановления поврежденных файлов формата *.bz2*.

НЕ ПРЕВРАЩАЙТЕСЬ В ОДЕРЖИМЫХ МАНИЕЙ СЖАТИЯ

Мне иногда приходится видеть, как кто-то пытается сжать файл, уже сжатый с применением эффективного алгоритма сжатия, выполняя нечто подобное:

```
$ gzip picture.jpg
```

Это напрасная трата времени и дискового пространства! Если применить процедуру сжатия к уже сжатому файлу, зачастую получается файл большего размера. Это объясняется тем, что все методики сжатия добавляют в файл некоторую служебную информацию, описывающую сжатие. Если попытаться сжать файл, не содержащий избыточной информации, сжатие не приведет к экономии места, которая могла бы покрыть расходы на хранение служебной информации.

Архивирование файлов

Часто вместе со сжатием используется операция *архивирования*. Архивирование — это процесс сбора множества файлов и упаковывание их в один большой файл. Архивирование часто применяется как один из этапов создания резервных

копий системы. Оно также используется при перемещении старых данных из системы в некоторое долговременное хранилище.

tar

В мире программного обеспечения для Unix-подобных систем существует программа `tar` — классический инструмент для архивирования файлов. Ее имя, которое расшифровывается как *tape archive* (архив на магнитной ленте), указывает, что первоначально инструмент предназначался для создания архивов на магнитных лентах. Он до сих пор используется для решения этой традиционной задачи, но с неменьшим успехом поддерживает другие устройства хранения. Нам часто приходится видеть имена файлов с расширением *.tar* или *.tgz*, которые обозначают «простые» tar-архивы и архивы, сжатые с помощью `gzip` соответственно. Архив может состоять из группы отдельных файлов, иерархий каталогов или и того и другого. Команда `tar` имеет следующий синтаксис:

```
tar режим[параметры] путь...
```

где под `режимом` подразумевается один из нескольких режимов работы, перечисленных в табл. 18.2 (здесь представлены не все параметры; полный список вы найдете на странице справочного руководства (man) для `tar`).

Таблица 18.2. Режимы команды tar

Режим	Описание
c	Создать архив из списка файлов и/или каталогов
x	Извлечь файлы из архива
r	Добавить указанный файл и/или каталог в конец архива
t	Вывести список содержимого архива

В программе `tar` используется немного непривычный способ определения параметров, поэтому рассмотрим несколько примеров ее использования. Для начала воссоздадим нашу песочницу, как мы это делали в предыдущей главе:

```
[me@linuxbox ~]$ mkdir -p playground/dir-{001..100}
[me@linuxbox ~]$ touch playground/dir-{001..100}/file-{A..Z}
```

Далее создадим архив всей песочницы:

```
[me@linuxbox ~]$ tar cf playground.tar playground
```

Эта команда создаст tar-архив с именем *playground.tar*, включающий всю иерархию каталогов песочницы. Как видите, режим и параметр `f`, который используется для определения имени tar-архива, можно объединять, и при этом не требуется использовать начальный дефис. Но имейте в виду, что режим всегда должен указываться первым, перед любыми параметрами. Посмотреть содержимое архива можно с помощью следующей команды:

```
[me@linuxbox ~]$ tar tf playground.tar
```

Для получения более подробного списка добавим параметр `v` (verbose — подробности):

```
[me@linuxbox ~]$ tar tvf playground.tar
```

Теперь извлечем содержимое архива в другой каталог. Для этого создадим новый каталог с именем *foo*, перейдем в него и извлечем содержимое tar-архива:

```
[me@linuxbox ~]$ mkdir foo
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground.tar
[me@linuxbox foo]$ ls
playground
```

Если внимательно исследовать содержимое *~/foo/playground*, можно заметить, что в результате распаковывания архива мы получили точные копии оригинальных файлов. Однако следует помнить, что если вы не действуете от имени суперпользователя, файлы и каталоги, извлеченные из архива, будут принадлежать пользователю, выполнившему восстановление, а не первоначальному их владельцу.

Другой интересной особенностью `tar` является способ обработки путей в архивах. По умолчанию используются относительные пути, а не абсолютные. Для этого программа `tar` просто удаляет начальный слеш во всех путях. Чтобы показать это, создадим снова наш архив, но на этот раз укажем абсолютный путь к архивируемому каталогу:

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ tar cf playground2.tar ~/playground
```

Как вы помните, командная оболочка заменит *~/playground* полным путем */home/me/playground* после нажатия клавиши ENTER, благодаря этому мы получим полный путь для нашей демонстрации. Далее извлечем архив, так же как прежде, и посмотрим, что из этого получилось:

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground2.tar
```

```
[me@linuxbox foo]$ ls
home   playground
[me@linuxbox foo]$ ls home
me
[me@linuxbox foo]$ ls home/me
playground
```

Как видите, здесь при извлечении архива каталог *home/me/playground* был воссоздан не в корневом, а в текущем рабочем каталоге *~/foo*, как было бы в случае с абсолютными путями. Это может показаться странным, но такое решение имеет свои преимущества: оно позволяет извлекать архивы в любое другое место, а не только в исходное. Повторив это упражнение с параметром, управляющим выводом дополнительных сообщений (v), можно получить более понятную картину происходящего.

Рассмотрим пусть и гипотетический, но все же имеющий практическую ценность пример использования tar. Представим, что нужно скопировать домашний каталог со всем его содержимым в другую систему и у нас имеется жесткий диск, подключаемый к порту USB, который можно использовать для переноса файлов. В современных системах Linux такие диски «как по волшебству» автоматически монтируются в каталог */media*. Допустим также, что подключаемый жесткий диск имеет том с именем *BigDisk*. Чтобы создать требуемый архив, выполним следующую команду:

```
[me@linuxbox ~]$ sudo tar cf /media/BigDisk/home.tar /home
```

После записи файла следует отмонтировать диск и подключить его ко второму компьютеру. И снова он автоматически монтируется в каталог */media/BigDisk*. Чтобы извлечь архив, выполните следующие команды:

```
[me@linuxbox2 ~]$ cd /
[me@linuxbox2 /]$ sudo tar xf /media/BigDisk/home.tar
```

Обратите внимание, что здесь сначала выполняется переход в каталог */*, чтобы извлечение производилось относительно корневого каталога, потому что все пути в архиве — относительные.

При распаковке архива можно ограничить количество извлекаемых данных. Например, можно извлечь из архива единственный файл:

```
tar xf archive.tar путь_к_файлу
```

Добавление в конец команды пути к файлу гарантирует извлечение только этого файла. Можно указать несколько путей. Обратите внимание, что путь к файлу должен быть полным относительным путем в архиве. Обычно в путях к файлам

нельзя использовать групповые символы; но GNU-версия `tar` (именно эта версия входит в состав большинства дистрибутивов Linux) поддерживает параметр `--wildcards`. В следующем примере используется файл *playground2.tar*, созданный выше:

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ tar xf ../playground2.tar --wildcards 'home/me/playground/dir-*/
file-A'
```

Эта команда извлечет только файлы, соответствующие указанному пути с групповым символом *dir-**.

Программа `tar` часто используется в сочетании с `find`. В следующем примере команда `find` используется для поиска файлов, подлежащих включению в архив:

```
[me@linuxbox ~]$ find playground -name 'file-A' -exec tar rf playground.tar '{}'
'+'
```

Здесь команда `find` отыскивает в каталоге *playground* все файлы с именем *file-A* и затем с помощью операции `-exec` вызывает `tar` в режиме добавления в конец (`r`), чтобы добавить найденные файлы в архив *playground.tar*.

Использование `tar` в сочетании с `find` предоставляет отличный способ *инкрементного резервного копирования* дерева каталогов или всей системы. Применяя `find` для поиска файлов, более новых, чем эталонный файл, определяющий отметку времени, можно создать архив, содержащий только более новые файлы, чем файлы предыдущего архива, при этом предполагается, что время последнего изменения эталонного файла будет изменяться сразу после создания архива.

Программа `tar` способна также использовать стандартный ввод и стандартный вывод. Например:

```
 [me@linuxbox foo]$ cd
[me@linuxbox ~]$ find playground -name 'file-A' | tar cf - --files-from=- | gzip >
playground.tgz
```

Здесь программа `find` создает список файлов и передает его по конвейеру программе `tar`. Когда программе `tar` передается имя файла `-` (дефис), под ним подразумевается стандартный ввод или стандартный вывод, в зависимости от контекста. (Кстати, соглашение об использовании дефиса (`-`) для представления стандартного ввода/вывода используется также многими другими программами.) Параметр `--files-from` (который можно заменить эквивалентным параметром `-T`) заставляет `tar` читать список путей из файла, а не из командной строки. Наконец, архив, произведенный программой `tar`, передается по конвейеру программе `gzip`, чтобы в результате получить сжатый архив *playground.tgz*. Расширение *.tgz* по

общепринятому соглашению используется для tar-архивов, сжатых программой `gzip`. В некоторых случаях используется расширение *.tar.gz*.

В примере, приведенном выше, для сжатия архива использовалась внешняя программа `gzip`, однако современные GNU-версии `tar` поддерживают возможность gzip- и bzip2-сжатия своими встроенными средствами, для чего служат параметры `z` и `j` соответственно. Взяв за основу предыдущий пример, его можно упростить, как показано ниже:

```
[me@linuxbox ~]$ find playground -name 'file-A' | tar czf playground.tgz -T -
```

Если, напротив, понадобится создать архив, сжатый в формате `bzip2`, это можно сделать так:

```
[me@linuxbox ~]$ find playground -name 'file-A' | tar cjf playground.tbz -T -
```

Произведя простую замену параметра сжатия `z` на `j` (и изменив расширение выходного файла на *.tbz*, указывающее, что для сжатия использовался алгоритм *bzip2*), мы задействовали bzip2-сжатие.

Другой интересный пример использования поддержки стандартного ввода и вывода командой `tar` связан с передачей файлов между системами по сети. Представьте, что имеется две машины, действующие под управлением Unix-подобных систем и имеющие программы `tar` и `ssh`. В этом случае можно организовать передачу каталога из удаленной системы (с именем `remote-sys` в этом примере) в локальную:

```
[me@linuxbox ~]$ mkdir remote-stuff
[me@linuxbox ~]$ cd remote-stuff
[me@linuxbox remote-stuff]$ ssh remote-sys 'tar cf - Documents' | tar xf -
me@remote-sys's password:
[me@linuxbox remote-stuff]$ ls
Documents
```

Здесь мы скопировали каталог *Documents* из удаленной системы `remote-sys` в каталог с именем *remote-stuff* в локальной системе. Как это получилось? Во-первых, мы запустили программу `tar` в удаленной системе с помощью команды `ssh`. Как вы наверняка помните, `ssh` позволяет выполнить программу на удаленном компьютере в сети и «увидеть» результат в локальной системе — стандартный вывод, полученный в удаленной системе, пересылается в локальную систему для обзора. Мы воспользовались этой особенностью и заставили `tar` создать архив (режим `c`) и вывести его не в файл, а в стандартный вывод (параметр `f` с дефисом в качестве аргумента), вследствие чего архив передается через шифрованный туннель, созданный программой `ssh`, локальной системе. В локальной системе мы вызвали `tar` с целью распаковать архив (режим `x`), полученный со стандартного ввода (все тот же параметр `f` с дефисом в качестве аргумента).

zip

Программа `zip` одновременно является и инструментом сжатия, и архиватором. Формат файлов, используемый программой, знаком пользователям Windows — программа читает и создает файлы с расширением *.zip*. Однако в Linux чаще других используется программа сжатия `gzip`, а второе место занимает `bzip2`. Пользователи Linux используют `zip` в основном для обмена файлами с системами Windows, а не как основной инструмент сжатия и архивирования.

В простейшем случае программа `zip` имеет следующий синтаксис:

```
zip параметры сжатый_файл файл...
```

Например, ниже показано, как создать zip-архив нашей песочницы:

```
[me@linuxbox ~]$ zip -r playground.zip playground
```

Без параметра `-r` (отвечает за рекурсивный обход каталогов) в архив будет включен только каталог *playground* (без своего содержимого). Расширение *.zip* добавляется к имени выходного файла автоматически, а мы включили его в пример для наглядности.

В процессе создания zip-архива программа `zip` обычно выводит последовательность сообщений, как показано ниже:

```
adding: playground/dir-020/file-Z (stored 0%)
adding: playground/dir-020/file-Y (stored 0%)
adding: playground/dir-020/file-X (stored 0%)
adding: playground/dir-087/ (stored 0%)
adding: playground/dir-087/file-S (stored 0%)
```

Эти сообщения показывают состояние каждого файла, добавленного в архив. `zip` добавляет файлы в архив, используя один из двух методов: либо «store» (простое сохранение) — без сжатия, как в примере, приведенном выше, либо «deflate» — со сжатием. Числовое значение, следующее за названием метода добавления, указывает достигнутую степень сжатия. Поскольку в нашей песочнице хранятся только пустые файлы, сжатие их содержимого не производится.

Извлечение содержимого из zip-архива выполняется просто — с помощью программы `unzip`:

```
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ unzip ../playground.zip
```

Одно важное отличие zip (от tar) состоит в том, что если указанный архив существует, он дополняется, а не замещается. То есть существующий архив сохраняется, новые файлы добавляются в него, а существующие — замещаются.

Программа unzip позволяет выводить информацию о файлах и выборочно извлекать их, достаточно только передать ей имя интересующего нас файла:

```
[me@linuxbox ~]$ unzip -l playground.zip playground/dir-087/file-Z
Archive:  ./playground.zip
  Length     Date   Time   Name
 --------    ----   ----   ----
        0  10-05-18 09:25  playground/dir-087/file-Z
 --------                  -------
        0                  1 file
[me@linuxbox ~]$ cd foo
[me@linuxbox foo]$ unzip ../playground.zip playground/dir-087/file-Z
Archive: ../playground.zip
replace playground/dir-087/file-Z? [y]es, [n]o, [A]ll, [N]one, [r]ename: y
  extracting: playground/dir-087/file-Z
```

При наличии параметра -l программа unzip просто выведет информацию о содержимом архива, не извлекая файл. Если имя файла (или файлов) не указано, unzip выведет список всех файлов в архиве. Для получения более подробной информации следует добавить параметр -v. Обратите внимание, что когда при извлечении из архива обнаруживается конфликт с существующим файлом, перед его заменой у пользователя запрашивается разрешение.

Подобно программе tar, zip может использовать стандартный ввод и вывод, хотя реализация этой возможности имеет меньшую практическую ценность. С помощью параметра -@ программе zip по конвейеру передается список имен файлов:

```
[me@linuxbox foo]$ cd
[me@linuxbox ~]$ find playground -name "file-A" | zip -@ file-A.zip
```

Здесь команда find генерирует список файлов, соответствующих проверке -name "file-A", и передает его по конвейеру команде zip, которая затем создает архив *file-A.zip* с выбранными файлами.

zip также поддерживает запись результатов своей работы в стандартный вывод, но эта особенность имеет ограниченное применение, потому что очень немногие программы способны работать с форматом zip. К сожалению, программа unzip не принимает входные данные со стандартного ввода. Это препятствует совместному использованию zip и unzip для копирования файлов по сети, как это возможно с программой tar.

`zip`, в свою очередь, способна принимать данные со стандартного ввода, поэтому ее можно использовать для сжатия вывода других программ:

```
[me@linuxbox ~]$ ls -l /etc/ | zip ls-etc.zip -
adding: - (deflated 80%)
```

В этом примере вывод команды `ls` передается по конвейеру программе `zip`. Так же как `tar`, `zip` интерпретирует завершающий дефис как требование «использовать стандартный ввод вместо файла».

Программа `unzip` позволяет направить ее результаты в стандартный вывод, для чего следует передать параметр `-p` (pipe — конвейер):

```
[me@linuxbox ~]$ unzip -p ls-etc.zip | less
```

Мы затронули лишь самые основные возможности программ `zip` и `unzip`. Обе они имеют множество параметров, придающих им большую гибкость, хотя некоторые из них допустимы только для определенных платформ. Для обеих программ, `zip` и `unzip`, имеются подробные страницы справочного руководства (man) с множеством полезных примеров; однако главное назначение этих программ — обмен файлами с системами Windows, а не сжатие и архивирование файлов в Linux, где большей популярностью пользуются `tar` и `gzip`.

Синхронизация файлов и каталогов

В задачах резервного копирования систем широко используется стратегия синхронизации одного или нескольких каталогов с другими каталогами, находящимися в локальной системе (обычно на некотором извлекаемом устройстве) или в удаленной. Можно, к примеру, создать локальную копию веб-сайта, находящегося в разработке, и синхронизировать ее время от времени с «рабочей» копией на удаленном веб-сервере.

В мире Unix-подобных систем для решения этой задачи широко используется инструмент `rsync`. Эта программа синхронизует локальные и удаленные каталоги, используя протокол *rsync remote-update* (протокол удаленного обновления rsync), который позволяет `rsync` быстро обнаруживать различия между двумя каталогами и копировать минимальный объем данных, необходимый для синхронизации. Это делает программу `rsync` быстрой и экономичной по сравнению с другими программами копирования.

Программа `rsync` имеет следующий синтаксис:

```
rsync параметры источник приемник
```

где роль источника и приемника могут играть:

- локальный файл или каталог;
- удаленный файл или каталог в форме *[пользователь@]хост:путь*;
- удаленный сервер rsync, определяемый идентификатором URI *rsync://[пользователь@]хост[:порт]/путь*.

Обратите внимание, что либо источник, либо приемник должен находиться в локальной системе. Копирование из удаленной системы в удаленную систему не поддерживается.

Давайте попробуем синхронизировать несколько локальных файлов. Сначала очистим наш каталог *foo*:

```
[me@linuxbox ~]$ rm -rf foo/*
```

Далее синхронизируем каталог *playground* с соответствующей копией в *foo*:

```
[me@linuxbox ~]$ rsync -av playground foo
```

Мы добавили два параметра: `-a` (для архивирования — обеспечивает рекурсивный обход и сохранение атрибутов файлов) и `-v` (подробный вывод), чтобы *отразить* каталог *playground* в каталог *foo*. В процессе выполнения команды можно просматривать список копируемых файлов и каталогов. В конце программа выведет итоговое сообщение, как показано ниже, включающее общий объем скопированных данных:

```
sent 135759 bytes   received 57870 bytes   387258.00 bytes/sec
total size is 3230   speedup is 0.02
```

Если теперь запустить команду еще раз, результат будет другой:

```
[me@linuxbox ~]$ rsync -av playgound foo
building file list ... done

 sent 22635 bytes   received 20 bytes  45310.00 bytes/sec
total size is 3230   speedup is 0.14
```

Обратите внимание на отсутствие списка файлов. Это объясняется тем, что программа `rsync` не обнаружила различий между *~/playground* и *~/foo/playground* и поэтому ничего не скопировала. Если теперь изменить файл в *playground* и запустить `rsync` еще раз, она обнаружит изменившийся файл и скопирует только его.

```
[me@linuxbox ~]$ touch playground/dir-099/file-Z
[me@linuxbox ~]$ rsync -av playground foo
building file list ... done
```

```
242 Глава 18. Архивация и резервное копирование
playground/dir-099/file-Z
sent 22685 bytes   received 42 bytes   45454.00 bytes/sec
total size is 3230   speedup is 0.14
```

Как видите, rsync обнаружила изменения и скопировала только изменившийся файл.

Есть одна маленькая, но важная деталь, связанная с определением источника в команде rsync. Пусть у нас есть два каталога:

```
[me@linuxbox ~]$ ls
source        destination
```

Каталог *source* содержит один файл с именем *file1*, а каталог *destination* пуст. Если выполнить копирование *source* в *destination* так:

```
[me@linuxbox ~]$ rsync source destination
```

rsync скопирует каталог *source* в *destination*:

```
[me@linuxbox ~]$ ls destination
source
```

Но если в имя каталога *source* добавить завершающий слеш /, rsync скопирует только содержимое каталога *source*, но не сам каталог:

```
[me@linuxbox ~]$ rsync source/ destination
[me@linuxbox ~]$ ls destination
file1
```

Это удобно, когда требуется скопировать только содержимое каталога, без создания еще одного уровня в дереве каталогов приемника. Этот трюк действует подобно *source/**, но, в отличие от последнего, обеспечит копирование всего содержимого каталога-источника, включая скрытые файлы.

В качестве практического примера представьте воображаемый внешний жесткий диск, использовавшийся выше с командой tar. Если после подключения такого диска к системе он снова будет смонтирован в каталог */media/BigDisk*, мы сможем выполнить первое резервное копирование системы, сначала создав каталог */backup* на внешнем устройстве, а затем вызвав rsync для копирования наиболее важных компонентов системы на внешнее устройство:

```
[me@linuxbox ~]$ mkdir /media/BigDisk/backup
[me@linuxbox ~]$ sudo rsync -av --delete /etc /home /usr/local /media/BigDisk/
backup
```

В этом примере мы скопировали каталоги */etc*, */home* и */usr/local* из нашей системы на воображаемый внешний диск. Мы добавили параметр `--delete`, чтобы удалить файлы, которые могут присутствовать на устройстве с резервной копией, но отсутствовать на устройстве-источнике (этот параметр не нужен при создании резервной копии в первый раз, но является полезным дополнением в последующих операциях копирования). Периодическое повторение процедуры подключения внешнего диска и запуск этой команды `rsync` является неплохим (хотя и не идеальным) способом сохранения резервной копии небольшой системы. Конечно, здесь также могло бы пригодиться создание псевдонима. Определим псевдоним и добавим его в свой файл *.bashrc*, чтобы обеспечить возможность быстрого резервного копирования:

```
alias backup='sudo rsync -av --delete /etc /home /usr/local /media/BigDisk/backup'
```

Теперь, чтобы выполнить всю работу, достаточно просто подключить внешний диск и ввести команду `backup`.

Использование rsync для копирования по сети

Одно из самых больших достоинств `rsync` — возможность копирования файлов по сети, об этом нам «говорит» буква `r` в названии `rsync`, что означает remote (удаленная). Удаленную синхронизацию можно выполнить одним из двух способов. Первый можно использовать с удаленными системами, где установлена `rsync` и программа удаленной командной оболочки, такая как `ssh`. Допустим, что в локальной сети имеется другая система с огромным объемом дискового пространства, и мы хотели бы использовать эту систему для хранения резервной копии вместо внешнего диска. Если допустить, что в этой системе уже имеется каталог */backup*, куда можно было бы сохранить наши файлы, мы могли бы выполнить резервное копирование так:

```
[me@linuxbox ~]$ sudo rsync -av --delete --rsh=ssh /etc /home /usr/local
remotesys:/backup
```

Мы внесли два изменения в команду, чтобы обеспечить копирование по сети. Во-первых, добавили параметр `--rsh=ssh`, который требует от `rsync` использовать в качестве удаленной командной оболочки программу `ssh`. Благодаря этому для передачи данных из локальной системы в удаленную мы можем использовать шифрованный туннель SSH. Во-вторых, мы добавили имя удаленного узла (в данном примере *remote-sys*) перед именем удаленного каталога.

Второй способ использования `rsync` для синхронизации файлов по сети заключается в использовании *сервера rysnc*. `rsync` можно настроить на работу в режиме демона, принимающего входящие запросы на синхронизацию. Этот прием

часто используется для зеркалирования удаленных систем. Например, компания Red Hat Software поддерживает огромный репозиторий программных пакетов, разрабатываемых для ее дистрибутива Fedora. Для специалистов, занимающихся тестированием программного обеспечения, очень удобно иметь зеркало этой коллекции в ходе этапа тестирования, предшествующего этапу выпуска дистрибутива. Поскольку файлы в репозитории обновляются достаточно часто (порой по нескольку раз в день), неплохо было бы организовать периодическую синхронизацию локального зеркала вместо копирования всего объема репозитория. Один из таких репозиториев хранится в университете Georgia Tech; мы могли бы создать его зеркало с помощью локальной программы `rsync` и сервера `rsync` в Georgia Tech:

```
[me@linuxbox ~]$ mkdir fedora-devel
[me@linuxbox ~]$ rsync -av --delete rsync://archive.linux.duke.edu/fedora/
linux/development/rawhide/Everything/x86_64/os/ fedora-devel
```

В этом примере мы использовали идентификатор URI удаленного сервера `rsync`, включающий протокол (*rsync://*), имя удаленного узла (*archive.linux.duke.edu*) и путь к репозиторию.

Заключение

Мы рассмотрели распространенные программы сжатия и архивирования, используемые в Linux и в других Unix-подобных операционных системах. Для архивирования файлов в Unix-подобных системах чаще других используется комбинация `tar`/`gzip`, тогда как команды `zip`/`unzip` в основном используются для организации обмена данными с системами Windows. Наконец, мы познакомились с программой `rsync` (моя любимица) — очень эффективным инструментом синхронизации файлов и каталогов между системами.

19 Регулярные выражения

В следующих нескольких главах мы познакомимся с инструментами для работы с текстом. Как вы уже знаете, текстовые данные играют важную роль в Unix-подобных системах, таких как Linux. Но прежде чем переходить к изучению возможностей этих инструментов, необходимо познакомиться с технологией, которая часто ассоциируется с самыми сложными случаями использования этих инструментов — *регулярными выражениями*.

Знакомясь со свойствами и особенностями командной строки, мы уже встречали некоторые по-настоящему таинственные свойства и команды, такие как механизмы подстановки и экранирования, короткие комбинации клавиш и история команд, не говоря уже о редакторе `vi`. Регулярные выражения продолжают этот список и являются (пожалуй) самым загадочным из всех инструментов. Это не означает, что время на их изучение будет потрачено впустую. Как раз наоборот. Хорошее понимание регулярных выражений позволит вам творить настоящие чудеса, хотя истинная их ценность поначалу может быть и не очевидна.

Что такое регулярные выражения?

Регулярные выражения — это всего лишь символическая форма записи, используемая для идентификации шаблонов в тексте. Они, до определенной степени, напоминают групповые символы, используемые командной оболочкой для выбора соответствующих файлов и путей, но в более широком масштабе. Регулярные выражения поддерживаются многими инструментами командной строки и большинством языков программирования, чтобы упростить решение задач, связанных с обработкой текста. Однако проблема в том, что не все регулярные выражения одинаковы; разные инструменты и языки программирования используют собственные «диалекты» регулярных выражений. Для целей нашего обсуждения мы

ограничимся регулярными выражениями, как они определены в стандарте POSIX (и поддерживаются большинством инструментов командной строки) в противоположность многим языкам программирования (особенно это относится к Perl), где используются более широкие и богатые формы записи.

grep

При работе с регулярными выражениями мы в основном будем использовать нашу старую добрую приятельницу — программу `grep`. Название *grep* в действительности произошло от фразы «global regular expression print» (глобальный поиск с помощью регулярного выражения и вывод), то есть, как видите, `grep` имеет некоторое отношение к регулярным выражениям. В сущности, `grep` просматривает текстовые файлы в поисках совпадений с указанным регулярным выражением и выводит в стандартный вывод все строки с такими совпадениями.

До сих пор мы передавали программе `grep` фиксированные строки, например:

```
[me@linuxbox ~]$ ls /usr/bin | grep zip
```

Эта команда выведет список всех файлов из каталога */usr/bin*, имена которых содержат подстроку *zip*.

Программа `grep` имеет следующий синтаксис:

```
grep [параметры] регулярное_выражение [файл...]
```

В табл. 19.1 перечислены наиболее часто используемые параметры `grep`.

Таблица 19.1. Параметры команды grep

Параметр	Длинный параметр	Описание
-i	--ignore-case	Игнорировать регистр символов. Требует не различать символы верхнего и нижнего регистров
-v	--invert-match	Инвертировать критерий. Обычно `grep` выводит строки с совпадениями. Этот параметр заставляет `grep` выводить строки, не содержащие совпадений
-c	--count	Вывести число совпадений (или «несовпадений») в присутствии параметра `-v` вместо самих текстовых строк

Параметр	Длинный параметр	Описание
-l	--files-with-matches	Вместо строк с совпадениями выводить только имена файлов с найденными строками
-L	--files-without-match	Действует подобно параметру -l, но выводит только имена файлов, где не найдено ни одного совпадения
-n	--line-number	В начале каждой строки с совпадением вывести ее номер в файле
-h	--no-filename	Подавить вывод имен файлов при поиске по нескольким файлам

Давайте создадим несколько текстовых файлов, чтобы наше исследование grep стало более предметным:

```
[me@linuxbox ~]$ ls /bin > dirlist-bin.txt
[me@linuxbox ~]$ ls /usr/bin > dirlist-usr-bin.txt
[me@linuxbox ~]$ ls /sbin > dirlist-sbin.txt
[me@linuxbox ~]$ ls /usr/sbin > dirlist-usr-sbin.txt
[me@linuxbox ~]$ ls dirlist*.txt
dirlist-bin.txt   dirlist-sbin.txt     dirlist-usr-sbin.txt
dirlist-usr-bin.txt
```

Ниже показано, как выполнить простой поиск в нашем списке файлов:

```
[me@linuxbox ~]$ grep bzip dirlist*.txt
dirlist-bin.txt:bzip2
dirlist-bin.txt:bzip2recover
```

В этом примере grep просматривает все перечисленные файлы в поисках строки *bzip* и находит два совпадения, оба в файле *dirlist-bin.txt*. Если бы нам достаточно было получить только имена файлов с совпадениями, а не сами совпадения, мы могли бы добавить параметр -l:

```
[me@linuxbox ~]$ grep -l bzip dirlist*.txt
dirlist-bin.txt
```

Напротив, получить список файлов, не содержащих совпадений, можно так:

```
[me@linuxbox ~]$ grep -L bzip dirlist*.txt
dirlist-sbin.txt
dirlist-usr-bin.txt
dirlist-usr-sbin.txt
```

Метасимволы и литералы

Несмотря на то что пока это не очевидно, во всех своих попытках поиска с помощью `grep` мы использовали регулярные выражения, хотя и очень простые. Регулярное выражение *bzip*, к примеру, означает, что ему соответствуют только строки в файлах, содержащие не менее четырех символов, и среди этих символов присутствуют символы *b*, *z*, *i* и *p*, следующие именно в таком порядке, и между ними отсутствуют какие-либо другие символы. Символы в строке *bzip* — это *литеральные символы*, то есть они соответствуют сами себе. Помимо литералов регулярные выражения могут содержать *метасимволы*, они используются для определения более сложных критериев сопоставления. К метасимволам регулярных выражений относятся следующие символы:

```
^ $ . [ ] { } - ? * + ( ) | \
```

Все остальные символы считаются литералами. Впрочем, в некоторых случаях символ «обратный слеш» используется для создания *метапоследовательностей*, а также для экранирования метасимволов, чтобы они могли интерпретироваться как литералы, а не как метасимволы.

ПРИМЕЧАНИЕ

Как видите, многие метасимволы регулярных выражений имеют также специальное значение для механизма подстановки командной оболочки. Поэтому, передавая регулярные выражения с метасимволами в виде аргументов командной строки, следует заключать их в кавычки, чтобы предотвратить попытки командной оболочки выполнить подстановку вместо них.

Любой символ

Первый метасимвол, который мы рассмотрим, — это символ «точка», соответствующий любому символу. Если включить его в регулярное выражение, он будет соответствовать любому символу в данной позиции. Например:

```
[me@linuxbox ~]$ grep -h '.zip' dirlist*.txt
bunzip2
bzip2
bzip2recover
gunzip
gzip
```

```
funzip
gpg-zip
preunzip
prezip
prezip-bin
unzip
unzipsfx
```

Здесь выполнен поиск в наших файлах совпадений с регулярным выражением *.zip*. В полученных результатах имеется пара важных моментов, которые необходимо отметить. Обратите внимание, что программа `zip` не была найдена. Это объясняется включением в регулярное выражение метасимвола «точка», увеличившим длину обязательного совпадения до четырех символов; так как в имени программы `zip` всего три символа, оно не было найдено. Кроме того, если бы в наших списках имелись имена файлов с расширением *.zip*, они также были бы найдены, потому что символ «точка» в расширении файла интерпретировался бы как «любой символ».

Якоря

Символ крышки (`^`) и знак доллара (`$`) в регулярных выражениях интерпретируются как *якоря*. Это означает, что в их присутствии совпадение с регулярным выражением возможно, только если совпадение будет найдено в начале строки (`^`) или в ее конце (`$`).

```
[me@linuxbox ~]$ grep -h '^zip' dirlist*.txt
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
[me@linuxbox ~]$ grep -h 'zip$' dirlist*.txt
gunzip
gzip
funzip
gpg-zip
preunzip
prezip
unzip
zip
[me@linuxbox ~]$ grep -h '^zip$' dirlist*.txt
zip
```

Здесь выполняется поиск в списке файлов строки *zip*, находящейся в начале строки, в конце строки и занимающей всю строку, от начала до конца. Обратите внимание, что регулярное выражение `^$` (начало и конец без каких-либо символов между ними) будет соответствовать пустым строкам.

В ПОМОЩЬ ЛЮБИТЕЛЯМ КРОССВОРДОВ

Даже наши ограниченные познания в области регулярных выражений могут принести некоторую пользу.

Моя супруга обожает разгадывать кроссворды и иногда просит меня помочь с ответом на какой-нибудь вопрос. Например: «Слово из пяти букв, третья *j*, последняя *r*, которое означает...» Подобные вопросы навели меня на размышления.

Знаете ли вы, что в вашей системе Linux имеется словарь? Загляните в каталог */usr/share/dict*, и вы обнаружите там один или несколько словарей. Файлы словарей, находящиеся в каталоге, — это обычные длинные списки слов, по одному в строке, упорядоченные по алфавиту. В моей системе файл *words* содержит больше 98 500 слов. Найти возможные ответы на вопрос в кроссворде можно с помощью следующей команды:

```
[me@linuxbox ~]$ grep -i '^..j.r$' /usr/share/dict/words
Major
major
```

Это регулярное выражение помогает найти в файле словаря все слова длиной в пять букв, где третья буква — *j* и последняя — *r*.

Выражения в квадратных скобках и классы символов

В дополнение к возможности описать в регулярном выражении совпадение с любым символом в заданной позиции, используя *выражения в квадратных скобках*, можно также описать совпадение с одним символом из определенного множества. Выражение в квадратных скобках помогает определить множество символов (включая символы, которые иначе интерпретировались бы как метасимволы), которые находятся в данной позиции. В следующем примере используется множество из двух символов, благодаря которому обнаруживаются соответствия с последовательностями *bzip* и *gzip*:

```
[me@linuxbox ~]$ grep -h '[bg]zip' dirlist*.txt
bzip2
```

```
bzip2recover
gzip
```

Множество может содержать любое число символов. Метасимволы, заключенные в квадратные скобки, теряют свое специальное значение. Лишь два метасимвола интерпретируются особым образом, но при этом они имеют иной смысл. Первый — символ крышки (^), который используется для обозначения отрицания; второй — дефис (-), который используется для обозначения диапазона символов.

Отрицание

Если сразу после открывающей квадратной скобки стоит символ крышки (^), остальные символы множества интерпретируются как недопустимые в данной позиции. Проверим это, изменив предыдущий пример:

```
[me@linuxbox ~]$ grep -h '[^bg]zip' dirlist*.txt
bunzip2
gunzip
funzip
gpg-zip
preunzip
prezip
prezip-bin
unzip
unzipsfx
```

Включив отрицание, мы получили список файлов, имена которых содержат последовательность *zip*, которой предшествует любой символ, кроме *b* или *g*. Обратите внимание, что файл *zip* не был найден. Символ отрицания не отменяет необходимости присутствия символа в заданной позиции, он лишь требует, чтобы символ в этой позиции не принадлежал указанному множеству.

Символ крышки обозначает операцию отрицания, только если является первым символом в выражении в квадратных скобках; в противном случае он теряет свое специальное значение и превращается в обычный символ.

Традиционные диапазоны символов

Если необходимо сконструировать регулярное выражение, которое находило бы в наших списках все файлы с именами, начинающимися с заглавной буквы, это можно выполнить следующим образом:

```
[me@linuxbox ~]$ grep -h '^[ABCDEFGHIJKLMNOPQRSTUVWXZY]' dirlist*.txt
```

Достаточно просто поместить 26 букв в верхнем регистре в выражение в квадратных скобках. Но необходимость ввода всех этих символов вызывает некоторое беспокойство, поэтому предусмотрен другой способ:

```
[me@linuxbox ~]$ grep -h '^[A-Z]' dirlist*.txt
MAKEDEV
ControlPanel
GET
HEAD
POST
X
X11
Xorg
MAKEFLOPPIES
NetworkManager
NetworkManagerDispatcher
```

Мы сократили множество с 26 буквами до 3-символьного диапазона. Так можно выразить любой диапазон символов и даже несколько диапазонов, например, для поиска имен файлов, начинающихся с буквы или цифры:

```
[me@linuxbox ~]$ grep -h '^[A-Za-z0-9]' dirlist*.txt
```

Как следует из примеров, символ «дефис» получает в диапазонах специальное значение, поэтому возникает вопрос: как включить дефис в выражение в квадратных скобках, чтобы он интерпретировался как обычный символ? Для этого достаточно поставить его в начало выражения. Например:

```
[me@linuxbox ~]$ grep -h '[A-Z]' dirlist*.txt
```

Эта команда найдет все имена файлов, содержащие буквы верхнего регистра. С другой стороны, следующее выражение:

```
[me@linuxbox ~]$ grep -h '[-AZ]' dirlist*.txt
```

найдет все имена файлов, содержащие дефис, букву *A* или букву *Z*.

Классы символов POSIX

Традиционные диапазоны символов — простой и эффективный способ определения наборов символов. К сожалению, они могут использоваться не со всеми программами. Мы не испытывали никаких проблем с диапазонами, используя программу `grep`, но могли бы столкнуться с ними при использовании других программ.

Вернемся к главе 4, где демонстрировалось использование групповых символов для подстановки имен файлов. Там говорилось, что можно использовать диапазоны символов почти так же, как они используются в регулярных выражениях, но есть одна проблема:

```
[me@linuxbox ~]$ ls /usr/sbin/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]*
/usr/sbin/MAKEFLOPPIES
/usr/sbin/NetworkManagerDispatcher
/usr/sbin/NetworkManager
```

(В разных дистрибутивах будут получены разные списки файлов, а в некоторых даже пустой список. Эти результаты получены в Ubuntu.) Эта команда вернула ожидаемый результат — список имен файлов, начинающихся с заглавной буквы. Но следующая команда даст совершенно другой результат (здесь приведена только часть результатов):

```
[me@linuxbox ~]$ ls /usr/sbin/[A-Z]*
/usr/sbin/biosdecode
/usr/sbin/chat
/usr/sbin/chgpasswd
/usr/sbin/chpasswd
/usr/sbin/chroot
/usr/sbin/cleanup-info
/usr/sbin/complain
/usr/sbin/console-kit-daemon
```

В чем же причина? Для этой длинной истории имеется короткая версия.

Во времена, когда операционная система Unix только появилась на свет, был известен только один набор символов — ASCII, и этот факт нашел свое отражение в данной особенности. В ASCII первые 32 символа (с номерами 0–31) — это управляющие символы (такие, как табуляция, забой и возврат каретки). Следующие 32 (32–63) представляют печатаемые символы, включая большинство знаков пунктуации и цифры с нуля до девяти. Следующие 32 (с номерами 64–95) представляют буквы верхнего регистра и несколько знаков пунктуации. Последние 31 (с номерами 96–127) представляют буквы нижнего регистра и еще несколько знаков пунктуации. Опираясь на эту классификацию, системы, использующие набор ASCII, придерживались следующего порядка сопоставления:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

Этот порядок отличается от лексикографического, который выглядит так:

```
aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ
```

С ростом популярности Unix за пределами США возникла необходимость в поддержке символов, не входящих в алфавит американского английского. Таблица ASCII была расширена до использования 8-битных символов, и в нее добавились символы с номерами 128–255, используемые во многих других языках. Для поддержки этой возможности в стандарт POSIX было введено понятие *региона* (locale), определяющее выбор набора символов для конкретного географического региона. Узнать, какой язык настроен в вашей системе, можно с помощью команды:

```
[me@linuxbox ~]$ echo $LANG
en_US.UTF-8
```

При проверке этой настройки POSIX-совместимые приложения используют лексикографический порядок, а не порядок следования символов в наборе ASCII. Это объясняет поведение команд, рассмотренное выше. Когда диапазон символов `[A-Z]` интерпретируется в лексикографическом порядке, он включает все алфавитные символы, кроме символа *a* в нижнем регистре, — именно это объясняет полученный результат.

Для частичного решения этой проблемы стандарт POSIX предусматривает несколько классов символов, описывающих диапазоны символов. Они перечислены в табл. 19.2.

Таблица 19.2. Классы символов POSIX

Класс символов	**Описание**
`[:alnum:]`	Алфавитно-цифровые символы; эквивалент диапазона `[A-Za-z0-9]` в ASCII
`[:word:]`	То же, что и `[:alnum:]`, с дополнительным символом подчеркивания (_)
`[:alpha:]`	Алфавитные символы; эквивалент диапазона `[A-Za-z]` в ASCII
`[:blank:]`	Включает символы пробела и табуляции
`[:cntrl:]`	Управляющие символы ASCII; включает символы ASCII с кодами от 0 до 31 и 127
`[:digit:]`	Цифры от 0 до 9
`[:graph:]`	Отображаемые символы; включает символы ASCII с кодами от 33 до 126
`[:lower:]`	Символы нижнего регистра

<table>
<tr><th>Класс символов</th><th>Описание</th></tr>
<tr><td>[:punct:]</td><td>Знаки пунктуации; эквивалент класса [-!"#$%&'()*+,./:;<=>?@[\\\]_`{|}~] в ASCII</td></tr>
<tr><td>[:print:]</td><td>Печатаемые символы; Все символы из класса [:graph:] и пробел</td></tr>
<tr><td>[:space:]</td><td>Пробельные символы, включая пробел, табуляцию, возврат каретки, перевод строки, вертикальную табуляцию и перевод формата; эквивалент класса [\t\r\n\v\f] в ASCII</td></tr>
<tr><td>[:upper:]</td><td>Символы верхнего регистра</td></tr>
<tr><td>[:xdigit:]</td><td>Символы, используемые для представления шестнадцатеричных цифр; эквивалент класса [0-9A-Fa-f] в ASCII</td></tr>
</table>

Но даже наличие классов символов не дает удобного способа выражения частичных диапазонов, таких как `[A-M]`.

Используя классы символов, можно повторить пример с выводом содержимого каталога и посмотреть, насколько улучшился результат.

```
[me@linuxbox ~]$ ls /usr/sbin/[[:upper:]]*
/usr/sbin/MAKEFLOPPIES
/usr/sbin/NetworkManagerDispatcher
/usr/sbin/NetworkManager
```

Но не забывайте, что это не является примером использования регулярных выражений, скорее это пример того, как командная оболочка выполняет подстановку путей. Мы рассмотрели его лишь потому, что классы символов POSIX поддерживаются и там и там.

Простые и расширенные регулярные выражения POSIX

Как раз когда, казалось бы, проблема путаницы с диалектами регулярных выражений решена, обнаруживается, что стандарт POSIX также делит реализации регулярных выражений на два вида: *простые регулярные выражения* (*Basic Regular Expressions, BRE*) и *расширенные регулярные выражения* (*Extended Regular Expressions, ERE*). Особенности, рассматривавшиеся до сих пор, поддерживаются всеми POSIX-совместимыми приложениями и приложениями, реализующими BRE. Программа `grep` — одна из них.

ВОЗВРАТ К ТРАДИЦИОННОМУ ПОРЯДКУ СОРТИРОВКИ

Есть возможность вернуть систему к традиционному (ASCII) порядку сортировки, изменив значение переменной окружения `LANG`. Как было показано в предыдущем разделе, переменная `LANG` хранит название языка и набора символов, заданных в региональных настройках. Значение этой переменной первоначально определяется в момент, когда выбирается язык установки дистрибутива Linux.

Увидеть региональные настройки можно, выполнив команду `locale`:

```
[me@linuxbox ~]$ locale
LANG=en_US.UTF-8
LC_CTYPE="en_US.UTF-8"
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"
LC_COLLATE="en_US.UTF-8"
LC_MONETARY="en_US.UTF-8"
LC_MESSAGES="en_US.UTF-8"
LC_PAPER="en_US.UTF-8"
LC_NAME="en_US.UTF-8"
LC_ADDRESS="en_US.UTF-8"
LC_TELEPHONE="en_US.UTF-8"
LC_MEASUREMENT="en_US.UTF-8"
LC_IDENTIFICATION="en_US.UTF-8"
LC_ALL=
```

Чтобы установить региональные настройки, обеспечивающие традиционное поведение системы Unix, присвойте переменной `LANG` значение `POSIX`:

```
[me@linuxbox ~]$ export LANG=POSIX
```

Имейте в виду, что в результате наших действий система будет использовать набор символов американского английского (точнее, ASCII), поэтому подумайте, действительно ли это то, что вам нужно.

Эти изменения можно сделать постоянными, добавив следующую строку в файл *.bashrc*:

```
export LANG=POSIX
```

Чем различаются BRE и ERE? Различия касаются наборов метасимволов. В диалекте BRE распознаются следующие метасимволы:

```
^ $ . [ ] *
```

Все остальные считаются литералами. В ERE во множество метасимволов (с соответствующими им функциями) добавляются:

```
( ) { } ? + |
```

POSIX

На протяжении 1980-х система Unix обрела популярность как коммерческая операционная система, но до 1988-го в мире Unix царила полная анархия. Многие производители компьютеров лицензировали исходный код Unix у ее создателя — компании AT&T и поставляли разные версии операционной системы вместе со своими машинами. Однако в стремлении к дифференциации продуктов каждый производитель добавлял свои, патентованные изменения и расширения. В результате значительно ухудшилась совместимость программного обеспечения. Как обычно, производители пытались играть в игру, победой в которой было «замыкание» клиентов на конкретном производителе. Этот период истории Unix ныне известен как *Балканизация* (Balkanization).

В середине 1980-х институт инженеров электроники и электротехники (Institute of Electrical and Electronics Engineers, IEEE) начал разработку единого пакета стандартов, которые должны были определить особенности работы системы Unix (и Unix-подобных). Эти стандарты, формально известные как IEEE 1003, определяют прикладные программные интерфейсы (Application Programming Interface, API), командную оболочку и утилиты, которые должны присутствовать в стандартной Unix-подобной системе. Название POSIX, сокращенное от «Portable Operating System Interface» (интерфейс переносимой операционной системы, где буква *X* добавлена для лучшего звучания), было предложено Ричардом Столлманом (да, тем самым Ричардом Столлманом) и принято IEEE.

Однако (что самое интересное) символы `( ) { }` интерпретируются в BRE как метасимволы, *если* они экранированы символом «обратный слеш», тогда как в ERE присутствие обратного слеша перед этими же метасимволами превращает их в литералы.

Поскольку далее в этой главе мы рассмотрим особенности, являющиеся частью ERE, необходимо использовать другую версию `grep`. Традиционно диалект ERE поддерживался программой `egrep`, но GNU-версия `grep` также поддерживает расширенные регулярные выражения при вызове ее с параметром `-E`.

Чередование

Первой особенностью расширенных регулярных выражений, которую мы обсудим, будет *чередование* (alternation, или *выражение выбора*) — оно позволяет выбирать совпадение с одним из нескольких выражений. Так же как выражения в квадратных скобках позволяют одному символу соответствовать множеству

указанных символов, чередование позволяет находить совпадение с множеством строк или других регулярных выражений.

Для демонстрации воспользуемся комбинацией команд `grep` и `echo`. Сначала попробуем выполнить простое сопоставление строк:

```
[me@linuxbox ~]$ echo "AAA" | grep AAA
AAA
[me@linuxbox ~]$ echo "BBB" | grep AAA
[me@linuxbox ~]$
```

Достаточно простой пример, в котором мы передаем по конвейеру вывод команды `echo` на ввод `grep` и видим результат. Если обнаруживается совпадение, мы видим вывод; если совпадение отсутствует, ничего не выводится.

Теперь добавим чередование, обозначаемое метасимволом вертикальной черты:

```
[me@linuxbox ~]$ echo "AAA" | grep -E 'AAA|BBB'
AAA
[me@linuxbox ~]$ echo "BBB" | grep -E 'AAA|BBB'
BBB
[me@linuxbox ~]$ echo "CCC" | grep -E 'AAA|BBB'
[me@linuxbox ~]$
```

Здесь мы видим регулярное выражение `'AAA|BBB'`, которое означает «совпадение со строкой `AAA` или со строкой `BBB`». Так как это расширенная особенность, мы добавили в команду `grep` параметр `-E` (вместо этого можно было бы использовать программу `egrep`) и заключили регулярное выражение в кавычки, чтобы предотвратить интерпретацию командной оболочкой символа вертикальной черты как оператора конвейера. В чередовании может быть более двух вариантов:

```
[me@linuxbox ~]$ echo "AAA" | grep -E 'AAA|BBB|CCC'
AAA
```

Для объединения с другими элементами регулярного выражения чередование можно заключать в круглые скобки `()`:

```
[me@linuxbox ~]$ grep -Eh '^(bz|gz|zip)' dirlist*.txt
```

Этому выражению будут соответствовать имена файлов из наших списков, начинающиеся с *bz*, *gz* или *zip*. Если отбросить круглые скобки, смысл регулярного выражения изменится, и ему будут соответствовать имена, начинающиеся с *bz* или содержащие *gz* или *zip*:

```
[me@linuxbox ~]$ grep -Eh '^bz|gz|zip' dirlist*.txt
```

Квантификаторы

Расширенные регулярные выражения поддерживают несколько способов определения числа совпадений с элементом.

? — совпадение с элементом ноль или один раз

Этот квантификатор фактически означает: «совпадение с предыдущим элементом не обязательно». Представьте, что нужно проверить допустимость номера телефона, и предполагается, что номер допустим, если представлен в одной из двух форм:

- *(nnn) nnn-nnnn*;
- *nnn nnn-nnnn*.

Для проверки можно было бы использовать следующее регулярное выражение:

```
^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$
```

В этом выражении за круглыми скобками следуют знаки вопроса, указывающие, что скобки могут либо отсутствовать, либо присутствовать один раз. И снова, поскольку круглые скобки считаются метасимволами (в ERE), мы экранировали их обратными слешами, чтобы они интерпретировались как литералы.

Попробуем применить это выражение:

```
[me@linuxbox ~]$ echo "(555) 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)? [0-9]
                                         [0-9][0-9]$'
(555) 123-4567
[me@linuxbox ~]$ echo "555 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)? [0-9][0-9]
                                       [0-9]-[0-9][0-9][0-9][0-9]$'
555 123-4567
[me@linuxbox ~]$ echo "AAA 123-4567" | grep -E '^\(?[0-9][0-9][0-9]\)? [0-9][0-9]
                                       [0-9]-[0-9][0-9][0-9][0-9]$'
[me@linuxbox ~]$
```

Здесь регулярному выражению соответствуют обе формы записи номера телефона, но ему не соответствует номер, содержащий нецифровые символы.

* — совпадение с элементом ноль или более раз

Подобно метасимволу `?`, звездочка (`*`) обозначает необязательный элемент; однако, в отличие от знака вопроса (`?`), этот элемент может встречаться любое число раз, а не только единожды. Представьте, что нам нужно проверить, является ли строка предложением. Чтобы удовлетворять нашим требованиям, строка должна

начинаться с большой буквы, содержать любое число букв верхнего и нижнего регистра и пробелов и заканчиваться точкой. Для поиска совпадений с этим (очень приблизительным) определением предложения воспользуемся следующим регулярным выражением:

```
[[:upper:]][[:upper:][:lower:] ]*\.
```

Выражение состоит из трех элементов: выражение в квадратных скобках с классом символов `[:upper:]`, выражение в квадратных скобках с двумя классами символов, `[:upper:]` и `[:lower:]`, и пробелом, и точка, экранированная обратным слешем. Второй элемент сопровождается метасимволом *, поэтому в нашем предложении ему может соответствовать любое число букв верхнего и нижнего регистра и пробелов, следующих за первой буквой верхнего регистра:

```
[me@linuxbox ~]$ echo "This works." | grep -E '[[:upper:]][[:upper:][:lower:] ]*\.'
This works.
[me@linuxbox ~]$ echo "This Works." | grep -E '[[:upper:]][[:upper:][:lower:] ]*\.'
This Works.
[me@linuxbox ~]$ echo "this does not" | grep -E '[[:upper:]][[:upper:][:lower:] ]*\.'
[me@linuxbox ~]$
```

Первые два примера соответствуют выражению, а третий — нет, потому что в нем отсутствует обязательный первый символ верхнего регистра и завершающая точка.

+ — совпадение с элементом один или более раз

Метасимвол + действует почти так же, как *, но требует совпадения с предыдущим элементом не менее одного раза. Следующему регулярному выражению будут соответствовать только строки, состоящие из групп, насчитывающих один или несколько алфавитных символов и разделенных одиночными пробелами:

```
^([[:alpha:]]+ ?)+$
```

Опробуем его:

```
[me@linuxbox ~]$ echo "This that" | grep -E '^([[:alpha:]]+ ?)+$'
This that
[me@linuxbox ~]$ echo "a b c" | grep -E '^([[:alpha:]]+ ?)+$'
a b c
[me@linuxbox ~]$ echo "a b 9" | grep -E '^([[:alpha:]]+ ?)+$'
[me@linuxbox ~]$ echo "abc  d" | grep -E '^([[:alpha:]]+ ?)+$'
[me@linuxbox ~]$
```

Как видите, этому выражению не соответствует строка "a b 9", потому что она содержит неалфавитный символ; точно так же ему не соответствует строка "abc d", потому что между символами *c* и *d* в ней присутствует больше одного пробела.

{ } — совпадение с элементом определенное число раз

Метасимволы { и } используются, чтобы выразить минимальное и максимальное число обязательных совпадений. Эти числа можно представить четырьмя возможными способами, как показано в табл. 19.3.

Таблица 19.3. Определение числа совпадений

Спецификатор	Значение
{n}	Предыдущий элемент соответствует, если встречается точно n раз
{n,m}	Предыдущий элемент соответствует, если встречается не менее n и не более m раз
{n,}	Предыдущий элемент соответствует, если встречается n или более раз
{,m}	Предыдущий элемент соответствует, если встречается не более m раз

Вернемся к примеру с телефонными номерами и воспользуемся этим методом определения повторений, чтобы упростить исходное регулярное выражение

```
^\(?[0-9][0-9][0-9]\)? [0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]$
```

до

```
^\(?[0-9]{3}\)? [0-9]{3}-[0-9]{4}$
```

Опробуем его:

```
[me@linuxbox ~]$ echo "(555) 123-4567" | grep -E '^\(?[0-9]{3}\)? [0-9]{3}-
[0-9]{4}$'
(555) 123-4567
[me@linuxbox ~]$ echo "555 123-4567" | grep -E '^\(?[0-9]{3}\)?
                                       [0-9]{3}-[0-9]{4}$'
555 123-4567
[me@linuxbox ~]$ echo "5555 123-4567" | grep -E '^\(?[0-9]{3}\)? [0-9]{3}-
[0-9]{4}$'
[me@linuxbox ~]$
```

Как видите, измененная версия регулярного выражения успешно справляется с проверкой номеров, с круглыми скобками и без них, и отвергает неправильно оформленные номера.

Практические примеры применения регулярных выражений

Рассмотрим несколько уже знакомых команд и посмотрим, как они могут использовать регулярные выражения.

Проверка списка телефонов с помощью grep

В предыдущем примере мы брали телефонные номера по одному и проверяли правильность их оформления. На практике же часто приходится проверять списки телефонов, поэтому давайте создадим такой список. Для этого воспользуемся волшебной магией командной строки. Магией, потому что мы еще не знакомы с большинством команд, привлеченных для решения поставленной задачи, но не волнуйтесь — мы рассмотрим их в последующих главах. Вот это волшебство:

```
[me@linuxbox ~]$ for i in {1..10}; do echo "(${RANDOM:0:3}) ${RANDOM:0:3}-
${RANDOM:0:4}" >> phonelist.txt; done
```

Эта команда создаст файл с именем *phonelist.txt*, содержащий 10 телефонных номеров. Если повторить команду, она добавит в список еще 10 номеров. Также можно изменить число 10 ближе к началу команды, чтобы создать больше или меньше номеров. Однако если заглянуть в файл, можно заметить проблему:

```
[me@linuxbox ~]$ cat phonelist.txt
(232) 298-2265
(624) 381-1078
(540) 126-1980
(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
```

Некоторые номера оформлены неправильно, что очень хорошо для целей демонстрации их проверки с помощью `grep`.

Было бы полезно просканировать файл в поисках недопустимых номеров и вывести их.

```
[me@linuxbox ~]$ grep -Ev '^\([0-9]{3}\) [0-9]{3}-[0-9]{4}$' phonelist.txt
(292) 108-518
(129) 44-1379
[me@linuxbox ~]$
```

Здесь мы использовали параметр -v, чтобы обратить сопоставление и вывести только строки, не соответствующие указанному выражению. Само выражение содержит якорные метасимволы на обоих концах и тем самым гарантирует отсутствие дополнительных символов слева и справа от номера. Кроме того, в отличие от примера, приведенного выше, это выражение также требует обязательного наличия круглых скобок в номере.

Поиск необычных имен файлов с помощью find

Команда find поддерживает проверку, основанную на регулярном выражении. Существует одно важное обстоятельство, которое следует помнить, используя регулярные выражения в командах find и grep. Если grep выводит строку, содержащую совпадение с регулярным выражением, то find требует точного совпадения пути с регулярным выражением. В следующем примере команда find использует регулярное выражение для поиска путей к файлам, содержащим любые символы, не входящие в следующее множество:

```
[-_./0-9a-zA-Z]
```

В результате такого поиска можно выявить имена файлов и каталогов, содержащие пробелы и другие, потенциально вредные символы:

```
[me@linuxbox ~]$ find . -regex '.*[^-_./0-9a-zA-Z].*'
```

Из-за требования точного совпадения всего пути мы добавили элемент .* с обоих концов выражения, замещающий любое количество любых символов (в том числе и отсутствие символов). В середине выражения находится инвертированное выражение в квадратных скобках, содержащее множество символов, допустимых в именах файлов и каталогов.

Поиск файлов с помощью locate

Программа locate поддерживает простые (параметр --regexp) и расширенные (параметр --regex) регулярные выражения. Благодаря этому можно выполнять те же операции, что производились выше с файлами dirlist:

```
[me@linuxbox ~]$ locate --regex ,bin/(bz|gz|zip)'
/bin/bzcat
/bin/bzcmp
/bin/bzdiff
/bin/bzegrep
/bin/bzexe
/bin/bzfgrep
/bin/bzgrep
/bin/bzip2
/bin/bzip2recover
/bin/bzless
/bin/bzmore
/bin/gzexe
/bin/gzip
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

Использовав чередование, мы нашли пути, содержащие *bin/bz*, *bin/gz* или */bin/zip*.

Поиск текста в less и vim

`less` и `vim` поддерживают одинаковые способы поиска в тексте. Чтобы выполнить поиск, нажмите клавишу / и введите регулярное выражение. Воспользуемся программой `less`, чтобы просмотреть содержимое файла *phonelist.txt*:

```
[me@linuxbox ~]$ less phonelist.txt
```

Затем выполним поиск с применением выражения для проверки:

```
(232) 298-2265
(624) 381-1078
(540) 126-1980
(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
~
~
~
/^\([0-9]{3}\) [0-9]{3}-[0-9]{4}$
```

less выделит строки с совпадениями, что позволит сразу увидеть недопустимые номера:

```
(232) 298-2265
(624) 381-1078
(540) 126-1980
(874) 163-2885
(286) 254-2860
(292) 108-518
(129) 44-1379
(458) 273-1642
(686) 299-8268
(198) 307-2440
~
~
~
(END)
```

Редактор vim поддерживает только простые регулярные выражения, поэтому выражение для поиска должно выглядеть следующим образом:

```
/([0-9]\{3\}) [0-9]\{3\}-[0-9]\{4\}
```

Как видите, выражение практически то же самое; различия обусловлены лишь тем, что многие символы, которые в расширенной версии выражений считаются метасимволами, в простой версии интерпретируются как литералы. Они действуют как метасимволы, только если экранировать их символом «обратный слеш». В зависимости от конкретных настроек vim совпадения могут быть выделены или нет. Если совпадения не выделяются, попробуйте в командном режиме выполнить команду :hlsearch, чтобы активировать выделение результатов поиска.

ПРИМЕЧАНИЕ

В разных дистрибутивах vim может поддерживать или не поддерживать подсветку результатов поиска в тексте. В Ubuntu, например, по умолчанию включена сильно упрощенная версия vim. В таких системах необходимо установить более полную версию vim с помощью диспетчера пакетов.

Заключение

В этой главе мы рассмотрели несколько примеров использования регулярных выражений. Круг практического применения регулярных выражений можно

расширить еще больше, если задействовать их для поиска в других приложениях, поддерживающих такую возможность. Например, с их помощью можно выполнять поиск на страницах справочного руководства:

```
[me@linuxbox ~]$ cd /usr/share/man/man1
[me@linuxbox man1]$ zgrep -El 'regex|regular expression' *.gz
```

Программа `zgrep` реализует интерфейс к `grep`, позволяя читать сжатые файлы. В данном примере выполняется поиск в сжатых файлах первого раздела справочного руководства. Результатом этой команды будет список файлов, содержащих строку *regex* или *regular expression*. Как видите, регулярные выражения поддерживаются множеством программ.

Простые регулярные выражения обладают одной интересной особенностью, которую мы пропустили, — *обратными ссылками* (back references). Они будут рассматриваться в следующей главе.

20 Обработка текста

Все Unix-подобные операционные системы широко используют текстовые файлы для хранения данных разных типов. Этим объясняется такое большое разнообразие инструментов обработки текста. В этой главе мы рассмотрим программы, которые используются для выполнения самых разных манипуляций с текстом. В следующей главе мы продолжим знакомство со средствами обработки текста, уделив больше внимания программам форматирования текста перед печатью и программам, удовлетворяющим другие потребности человека.

В этой главе мы повторно рассмотрим уже знакомые программы и познакомимся с новыми:

`cat` — объединяет файлы и выводит их в стандартный вывод;

`sort` — сортирует строки из текстовых файлов;

`uniq` — сообщает о повторяющихся строках или удаляет их;

`cut` — удаляет фрагменты из каждой строки в файлах;

`paste` — выполняет слияние строк из файлов;

`join` — объединяет строки из двух файлов по общему полю;

`comm` — выполняет построчное сравнение двух сортированных файлов;

`diff` — выполняет построчное сравнение файлов;

`patch` — применяет diff-файл (файл с результатами сравнения командой `diff`) к оригиналу;

`tr` — перекодирует или удаляет символы;

`sed` — потоковый редактор для фильтрации и преобразования текста;

`aspell` — интерактивная программа проверки орфографии.

Области применения текста

К настоящему моменту мы познакомились с парой текстовых редакторов (`nano` и `vim`), рассмотрели несколько конфигурационных файлов и увидели вывод нескольких десятков команд, и все это в текстовом виде. А для чего еще используется текст? Как оказывается, много для чего.

Документы

Многие люди записывают документы в простом текстовом формате. Очевидно, достаточно удобно хранить простые заметки в небольших текстовых файлах, однако и большие документы также можно записывать в простом текстовом формате. Один из популярных подходов состоит в том, чтобы записать большой документ в текстовом формате и затем использовать *язык разметки* для описания форматирования конечного документа. Многие научные статьи написаны подобным способом, так как системы обработки текста на основе Unix были в числе первых, обеспечивших улучшенную поддержку типографического оформления, так необходимого авторам в технических дисциплинах.

Веб-страницы

Самым популярным в мире форматом электронных документов является, пожалуй, формат веб-страниц. Веб-страницы — это текстовые документы с разметкой *HTML* (Hypertext Markup Language — язык разметки гипертекста) или *XML* (Extensible Markup Language — расширяемый язык разметки), описывающей визуальный формат документа.

Электронная почта

Электронная почта является текстовой средой по своей природе. Даже нетекстовые вложения преобразуются в текстовое представление перед передачей. В этом можно убедиться, загрузив электронное письмо и просмотрев его с помощью `less`. Вы увидите, что письмо начинается с *заголовка*, описывающего отправителя письма и промежуточные серверы, принимавшие его в процессе доставки, за которым следует тело письма с его содержимым.

Вывод на принтер

В Unix-подобных системах данные выводятся на печать в простом текстовом виде, или, если страница содержит графику, она преобразуется в описание страницы

в текстовом формате, известном как *PostScript*, которое затем посылается программе, генерирующей графические точки для печати на бумаге.

Исходный код программ

Многие программы командной строки, имеющиеся в Unix-подобных системах, были созданы для поддержки системного администрирования и разработки программного обеспечения, и программы обработки текста не исключение. Многие из них предназначались для решения задач, связанных с разработкой программного обеспечения. Важность обработки текста для программистов объясняется тем, что любое программное обеспечение начинает свое существование как текст. *Исходный код*, часть программы, которую пишет программист, всегда имеет текстовый формат.

А вот и наши старые знакомые!

В главе 6 мы познакомились с некоторыми командами, способными принимать данные не только из аргументов командной строки, но и из стандартного ввода. В той главе мы очень коротко познакомились с ними, но сейчас пришло время более близкого знакомства — мы узнаем, как их можно использовать для обработки текста.

cat

Программа `cat` содержит множество интересных параметров. Многие из них используются для улучшенного отображения текстового содержимого. Примером может служить параметр `-A`, используемый для отображения непечатаемых символов в тексте. Иногда необходимо знать, имеются ли управляющие символы в просматриваемом тексте. Наиболее распространенными из них являются символы табуляции (в противоположность пробелам) и символы возврата каретки, часто представляющие концы строк в текстовых файлах, оформленных в стиле MS-DOS. Другим распространенным вариантом является файл, содержащий строки с завершающими пробелами.

Давайте создадим файл для экспериментов, используя `cat` как примитивный текстовый процессор. Для этого введем команду `cat` (указав файл для перенаправления вывода), а следом — наш текст, завершив строки нажатием клавиши `ENTER` и закончив все комбинацией `CTRL+D` — она сообщит программе `cat`, что достигнут конец файла. В этом примере мы ввели символ табуляции и добавили в конец строки несколько пробелов:

```
[me@linuxbox ~]$ cat > foo.txt
    The quick brown fox jumped over the lazy dog[1].
[me@linuxbox ~]$
```

Далее, вызовем `cat` с параметром `-A`, чтобы показать текст:

```
[me@linuxbox ~]$ cat -A foo.txt
^IThe quick brown fox jumped over the lazy dog.   $
[me@linuxbox ~]$
```

Как видите, символ табуляции в тексте представлен парой символов `^I`. Эта обычная форма записи означает «CTRL+I», то есть, как оказывается, — символ табуляции. Здесь также видно, что символ `$` отмечает истинный конец строки, помогая увидеть дополнительные пробелы в конце строки.

ТЕКСТ В MS-DOS И UNIX

Одна из причин, почему может появиться желание использовать `cat` для отображения непечатаемых символов в тексте, — необходимость определить присутствие символов возврата каретки. Откуда берутся скрытые возвраты каретки? Из DOS и Windows! В Unix и DOS концы строк в текстовых файлах оформляются по-разному. В Unix строки заканчиваются символом перевода строки (ASCII 10), тогда как в MS-DOS и ее наследницах для этой цели используется последовательность из возврата каретки (ASCII 13) и перевода строки.

Существует несколько способов преобразовать файлы из формата DOS в формат Unix. Во многих системах Linux имеются программы `unix2dos` и `dos2unix` для преобразования текстовых файлов в формат DOS и обратно. Однако если в вашей системе нет программы `dos2unix`, не волнуйтесь. Процесс преобразования текста из формата DOS в формат Unix очень прост — достаточно лишь удалить ненужные возвраты каретки. Это можно сделать с помощью пары программ, с которыми мы познакомимся позже в этой главе.

Программа `cat` имеет также параметры, используемые для изменения текста. Наиболее известными являются `-n`, добавляющий номера строк, и `-s`, подавляющий вывод множества пустых строк, идущих подряд. Давайте продемонстрируем их:

[1] Английский аналог фразы: «Съешь же ещё этих мягких французских булок, да выпей чаю», содержащей все буквы алфавита. В этой главе, чтобы избежать полного переписывания всех авторских примеров, мы будем работать с ней. — *Примеч. ред.*

```
[me@linuxbox ~]$ cat > foo.txt
The quick brown fox


jumped over the lazy dog.
[me@linuxbox ~]$ cat -ns foo.txt
     1  The quick brown fox
     2
     3  jumped over the lazy dog.
[me@linuxbox ~]$
```

В этом примере мы создали новую версию тестового файла *foo.txt*, содержащую две строки, разделенные двумя пустыми строками. После обработки текста командой `cat` с параметрами `-ns` одна пустая строка была удалена, а остальные строки пронумерованы. Хотя это не самая сложная обработка текста, все же это обработка.

sort

Программа `sort` сортирует содержимое стандартного ввода или одного или нескольких файлов, указанных в командной строке, и записывает результаты в стандартный вывод. Применив тот же прием, который мы использовали совместно с командой `cat`, можно продемонстрировать обработку стандартного ввода:

```
[me@linuxbox ~]$ sort > foo.txt
c
b
a
[me@linuxbox ~]$ cat foo.txt
a
b
c
```

После запуска команды мы ввели буквы *c*, *b* и *a*, а затем признак конца файла с помощью комбинации CTRL+D. Затем просмотрели получившийся файл и увидели, что строки в нем отсортированы.

Поскольку `sort` может принимать несколько файлов в аргументах командной строки, существует возможность *объединить* множество файлов в один отсортированный файл. Например, если у вас имеется три файла и вам нужно объединить их в один отсортированный файл, это можно выполнить следующим образом:

```
sort file1.txt file2.txt file3.txt > final_sorted_list.txt
```

Программа `sort` имеет несколько интересных параметров. Их неполный список приводится в табл. 20.1.

Таблица 20.1. Часто используемые параметры команды sort

Параметр	Длинный параметр	Описание
-b	--ignore-leading-blanks	По умолчанию сортировка выполняется с учетом содержимого всей строки, начиная с первого символа. Этот параметр заставляет sort пропустить начальные пробелы в строках и при сортировке начинать сравнение строк с первых не пробельных символов
-f	--ignore-case	Выполнять сортировку без учета регистра символов
-n	--numeric-sort	Выполнять сортировку, опираясь на числовые значения строк. Этот параметр позволяет осуществлять сортировку в соответствии с числовыми значениями, а не по алфавиту
-r	--reverse	Сортировать в обратном порядке. Строки в результате будут следовать в порядке убывания, а не возрастания
-k	--key=поле1[,поле2]	Сортировать по ключевым полям, начиная с поля1 и заканчивая полем2, а не по всей строке (поясняется ниже)
-m	--merge	Интерпретировать каждый аргумент, как имя предварительно отсортированного файла. Позволяет объединить несколько файлов в общий результат без выполнения дополнительной сортировки
-o	--output=файл	Записать результат сортировки не в стандартный вывод, а в указанный файл
-t	--field-separator=символ	Определяет символ, разделитель полей. По умолчанию в качестве разделителя используются пробелы и символы табуляции

Имена большинства параметров из представленных выше говорят сами за себя, однако некоторые требуют дополнительных пояснений. Прежде всего рассмотрим параметр -n, используемый для сортировки по числовым значениям. Этот параметр позволяет сортировать строки по их числовым значениям. Продемонстрировать действие этого параметра можно на примере сортировки результатов команды du, чтобы определить каталог, занимающий больший объем дискового пространства. Обычно команда du выводит результаты, отсортированные по именам каталогов:

```
[me@linuxbox ~]$ du -s /usr/share/* | head
252         /usr/share/aclocal
96          /usr/share/acpi-support
8           /usr/share/adduser
196         /usr/share/alacarte
344         /usr/share/alsa
8           /usr/share/alsa-base
12488       /usr/share/anthy
8           /usr/share/apmd
21440       /usr/share/app-install
48          /usr/share/application-registry
```

В этом примере мы передали результаты по конвейеру программе `head`, чтобы ограничить число результатов первыми 10 строками. Мы можем изменить эту команду, добавив сортировку по числовым значениям, чтобы получить 10 самых объемных каталогов:

```
[me@linuxbox ~]$ du -s /usr/share/* | sort -nr | head
509940    /usr/share/locale-langpack
242660    /usr/share/doc
197560    /usr/share/fonts
179144    /usr/share/gnome
146764    /usr/share/myspell
144304    /usr/share/gimp
135880    /usr/share/dict
76508     /usr/share/icons
68072     /usr/share/apps
62844     /usr/share/foomatic
```

С помощью параметров `n` и `r` мы получили сортировку по числовым значениям в обратном порядке, в результате наибольшие значения оказались в начале списка. Такой способ сортировки стал возможен, потому что числовые значения находятся в начале каждой строки. Но как быть, если потребуется отсортировать строки по числовым значениям, находящимся в середине строки, как, например, в результатах команды `ls -l`?

```
[me@linuxbox ~]$ ls -l /usr/bin | head
total 152948
-rwxr-xr-x 1 root   root      34824 2016-04-04 02:42 [
-rwxr-xr-x 1 root   root     101556 2007-11-27 06:08 a2p
-rwxr-xr-x 1 root   root      13036 2016-02-27 08:22 aconnect
-rwxr-xr-x 1 root   root      10552 2007-08-15 10:34 acpi
-rwxr-xr-x 1 root   root       3800 2016-04-14 03:51 acpi_fakekey
-rwxr-xr-x 1 root   root       7536 2016-04-19 00:19 acpi_listen
-rwxr-xr-x 1 root   root       3576 2016-04-29 07:57 addpart
-rwxr-xr-x 1 root   root      20808 2016-01-03 18:02 addr2line
-rwxr-xr-x 1 root   root     489704 2016-10-09 17:02 adept_batch
```

Забудем на время, что `ls` может сортировать свои результаты по размеру, и выполним сортировку с помощью `sort`:

```
[me@linuxbox ~]$ ls -l /usr/bin | sort -nr -k 5 | head
-rwxr-xr-x 1 root   root    8234216 2016-04-07 17:42 inkscape
-rwxr-xr-x 1 root   root    8222692 2016-04-07 17:42 inkview
-rwxr-xr-x 1 root   root    3746508 2016-03-07 23:45 gimp-2.4
-rwxr-xr-x 1 root   root    3654020 2016-08-26 16:16 quanta
-rwxr-xr-x 1 root   root    2928760 2016-09-10 14:31 gdbtui
-rwxr-xr-x 1 root   root    2928756 2016-09-10 14:31 gdb
-rwxr-xr-x 1 root   root    2602236 2016-10-10 12:56 net
-rwxr-xr-x 1 root   root    2304684 2016-10-10 12:56 rpcclient
-rwxr-xr-x 1 root   root    2241832 2016-04-04 05:56 aptitude
-rwxr-xr-x 1 root   root    2202476 2016-10-10 12:56 smbcacls
```

Программа `sort` часто вовлекается в обработку табличных данных, таких как результат команды `ls` выше. Если воспользоваться терминологией баз данных, об этой таблице можно сказать, что каждая строка — это *запись* и каждая запись состоит из множества *полей*, таких как атрибуты файла, счетчик ссылок, имя файла, размер файла и т. д. Программа `sort` способна обрабатывать поля по отдельности. Согласно той же терминологии баз данных, мы можем указать одно или несколько *ключевых полей*, которые должны использоваться как *ключи сортировки*. В примере, приведенном выше, мы добавили параметры `n` и `r`, чтобы выполнить сортировку по числовым значениям в порядке убывания, а также параметр `k` с аргументом `5`, чтобы указать, что сортировка должна выполняться по пятому полю.

Параметр `k` очень интересен и обладает множеством любопытных свойств, но прежде чем приступить к их изучению, поговорим о том, как `sort` определяет поля. Рассмотрим очень простой текстовый файл, содержащий единственную строку с именем автора этой книги:

```
William Shotts
```

По умолчанию `sort` «видит» в этой строке два поля. Первое поле содержит последовательность символов `William`, второе — последовательность символов `Shotts`, то есть пробельные символы (пробелы и символы табуляции) интерпретируются как разделители полей, и эти разделители включаются в поле при выполнении сортировки.

Взглянув еще раз на любую строку в выводе нашей команды `ls`, можно сказать, что она содержит восемь полей и пятое поле хранит размер файла:

```
-rwxr-xr-x 1 root root 8234216 2016-04-07 17:42 inkscape
```

Для следующей серии экспериментов возьмем файл с историей выпуска новых версий трех популярных дистрибутивов Linux в период с 2006 по 2008 год. Каждая

строка в файле содержит три поля: название дистрибутива, номер версии и дата выпуска в формате ММ/ДД/ГГГГ:

```
SUSE           10.2    12/07/2006
Fedora         10      11/25/2008
SUSE           11.0    06/19/2008
Ubuntu         8.04    04/24/2008
Fedora         8       11/08/2007
SUSE           10.3    10/04/2007
Ubuntu         6.10    10/26/2006
Fedora         7       05/31/2007
Ubuntu         7.10    10/18/2007
Ubuntu         7.04    04/19/2007
SUSE           10.1    05/11/2006
Fedora         6       10/24/2006
Fedora         9       05/13/2008
Ubuntu         6.06    06/01/2006
Ubuntu         8.10    10/30/2008
Fedora         5       03/20/2006
```

С помощью текстового редактора (например, `vim`) введите эти данные и сохраните в файле с именем *distros.txt*.

Далее попробуем отсортировать файл и посмотрим, что из этого получится:

```
[me@linuxbox ~]$ sort distros.txt
Fedora         10      11/25/2008
Fedora         5       03/20/2006
Fedora         6       10/24/2006
Fedora         7       05/31/2007
Fedora         8       11/08/2007
Fedora         9       05/13/2008
SUSE           10.1    05/11/2006
SUSE           10.2    12/07/2006
SUSE           10.3    10/04/2007
SUSE           11.0    06/19/2008
Ubuntu         6.06    06/01/2006
Ubuntu         6.10    10/26/2006
Ubuntu         7.04    04/19/2007
Ubuntu         7.10    10/18/2007
Ubuntu         8.04    04/24/2008
Ubuntu         8.10    10/30/2008
```

У нас это почти получилось. Единственная проблема возникла с сортировкой номеров версий Fedora. Так как в лексикографическом смысле 1 предшествует 5, версия 10 оказалась вверху, тогда как версия 9 — внизу.

Чтобы исправить эту ошибку, выполним сортировку по нескольким ключам. Итак, нам нужно выполнить сортировку по первому полю в алфавитном порядке, а затем по второму полю в числовом порядке. Программа `sort` позволяет указать в командной строке несколько параметров `-k`, чтобы можно было определить несколько ключей сортировки. В действительности в ключ можно включать диапазон полей. Если диапазон не определен (как в примерах, приведенных выше), `sort` использует в качестве ключа часть строки, начинающуюся с указанного поля и простирающуюся до конца строки.

Вот как выглядит синтаксис сортировки по нескольким ключам:

```
[me@linuxbox ~]$ sort --key=1,1 --key=2n distros.txt
Fedora          5       03/20/2006
Fedora          6       10/24/2006
Fedora          7       05/31/2007
Fedora          8       11/08/2007
Fedora          9       05/13/2008
Fedora          10      11/25/2008
SUSE            10.1    05/11/2006
SUSE            10.2    12/07/2006
SUSE            10.3    10/04/2007
SUSE            11.0    06/19/2008
Ubuntu          6.06    06/01/2006
Ubuntu          6.10    10/26/2006
Ubuntu          7.04    04/19/2007
Ubuntu          7.10    10/18/2007
Ubuntu          8.04    04/24/2008
Ubuntu          8.10    10/30/2008
```

Здесь для ясности использовались имена параметров в длинной форме, однако с тем же успехом можно было бы передать параметры `-k 1,1 -k 2n`. В аргументе для первого экземпляра параметра ключа мы указали диапазон полей, входящих в первый ключ. Так как сортировка должна выполняться только по первому полю, мы указали диапазон `1,1`, что означает: «начиная с поля 1 и заканчивая полем 1». Второму экземпляру мы передали аргумент `2n`, который означает: «ключом сортировки является второе поле, и сортировка выполняется в порядке числовых значений». В конце определения ключа допускаются однобуквенные имена параметров, они указывают на тип сортировки. Имена этих однобуквенных параметров совпадают с именами глобальных параметров программы `sort`: `b` (пропустить начальные пробелы), `n` (числовая сортировка), `r` (сортировка в обратном порядке) и т. д.

Третье поле в списке содержит дату в формате, неудобном для сортировки. В компьютере даты обычно приводятся к виду ГГГГ-ММ-ДД, что упрощает сортировку

в хронологическом порядке, но здесь используется американский формат ММ/ДД/ГГГГ. Как же тогда отсортировать этот список в хронологическом порядке?

К счастью, `sort` предоставляет такую возможность. Параметр `--key` позволяет определять *смещения* внутри полей, чтобы в качестве ключей можно было использовать части полей:

```
[me@linuxbox ~]$ sort -k 3.7nbr -k 3.1nbr -k 3.4nbr distros.txt
Fedora          10      11/25/2008
Ubuntu          8.10    10/30/2008
SUSE            11.0    06/19/2008
Fedora          9       05/13/2008
Ubuntu          8.04    04/24/2008
Fedora          8       11/08/2007
Ubuntu          7.10    10/18/2007
SUSE            10.3    10/04/2007
Fedora          7       05/31/2007
Ubuntu          7.04    04/19/2007
SUSE            10.2    12/07/2006
Ubuntu          6.10    10/26/2006
Fedora          6       10/24/2006
Ubuntu          6.06    06/01/2006
SUSE            10.1    05/11/2006
Fedora          5       03/20/2006
```

Добавив параметр `-k 3.7`, мы сообщили программе `sort`, что она должна использовать для сортировки ключ, начинающийся с седьмого символа в третьем поле, который соответствует началу года. Аналогично, параметры `-k 3.1` и `-k 3.4` определяют ключи сортировки по месяцу и дню месяца. Мы также добавили параметры `n` и `r`, чтобы выполнить числовую сортировку в обратном порядке. Параметр `b` добавлен для исключения начальных пробелов из поля с датой (число которых в разных строках отличается и тем самым влияет на результат сортировки).

В некоторых файлах в качестве разделителей используются символы, отличные от пробелов и символов табуляции; возьмем, к примеру, файл */etc/passwd*:

```
[me@linuxbox ~]$ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
```

Поля в этом файле отделяются друг от друга двоеточием (`:`). Можно ли отсортировать содержимое этого файла с использованием ключевых полей? Программа `sort` поддерживает параметр `-t` для определения символа — разделителя полей. Чтобы отсортировать содержимое файла *passwd* по седьмому полю (командная оболочка по умолчанию), используем такую команду:

```
[me@linuxbox ~]$ sort -t ':' -k 7 /etc/passwd | head
me:x:1001:1001:Myself,,,:/home/me:/bin/bash
root:x:0:0:root:/root:/bin/bash
dhcp:x:101:102::/nonexistent:/bin/false
gdm:x:106:114:Gnome Display Manager:/var/lib/gdm:/bin/false
hplip:x:104:7:HPLIP system user,,,:/var/run/hplip:/bin/false
klog:x:103:104::/home/klog:/bin/false
messagebus:x:108:119::/var/run/dbus:/bin/false
polkituser:x:110:122:PolicyKit,,,:/var/run/PolicyKit:/bin/false
pulse:x:107:116:PulseAudio daemon,,,:/var/run/pulse:/bin/false
```

Определив двоеточие как разделитель полей, мы смогли выполнить сортировку по седьмому полю.

uniq — выявление или удаление повторяющихся строк

В сравнении с `sort` программа `uniq` более легковесна. Она решает, казалось бы, тривиальную задачу. Когда ей передается сортированный файл (в том числе и стандартный ввод), она удаляет повторяющиеся строки и выводит результат в стандартный вывод. Она часто используется в сочетании с `sort` для удаления повторяющихся строк.

СОВЕТ

Даже притом, что `uniq` — инструмент, традиционный для Unix, который часто используется вместе с `sort`, GNU-версия `sort` поддерживает параметр `-u`, удаляющий повторяющиеся строки из сортированных результатов.

Давайте создадим текстовый файл для последующих экспериментов:

```
[me@linuxbox ~]$ cat > foo.txt
a
b
c
a
b
c
```

Не забудьте ввести CTRL+D, чтобы завершить ввод с клавиатуры. Если теперь применить uniq к нашему текстовому файлу, результат ничем не будет отличаться от оригинала; повторяющиеся записи никуда не исчезли:

```
[me@linuxbox ~]$ uniq foo.txt
a
b
c
a
b
c
```

Чтобы uniq действительно выполнила свою работу, исходные данные нужно сначала отсортировать:

```
[me@linuxbox ~]$ sort foo.txt | uniq
a
b
c
```

Это объясняется тем, что uniq удаляет повторяющиеся записи, только если они следуют друг за другом.

uniq имеет несколько параметров. Наиболее часто используемые из них перечислены в табл. 20.2.

Таблица 20.2. Часто используемые параметры команды uniq

Параметр	Длинный параметр	Описание
-c	--count	Вывести список повторяющихся строк, предваряя их числом найденных дубликатов
-d	--repeated	Вывести только повторяющиеся, не уникальные строки
-f n	--skip-fields=n	Пропустить n начальных полей в каждой строке. Деление на поля производится по пробельным символам, как в программе sort; однако, в отличие от sort, программа uniq не имеет параметра для настройки альтернативного разделителя полей
-i	--ignore-case	Сравнивать строки без учета регистра символов
-s n	--skip-chars=n	Пропустить n начальных символов в каждой строке
-u	--unique	Вывести только уникальные строки. Подразумевается по умолчанию

В следующем примере используется параметр `-c` программы `uniq` для определения числа повторяющихся строк в исходном текстовом файле:

```
[me@linuxbox ~]$ sort foo.txt | uniq -c
2 a
2 b
2 c
```

Нарезка и перетасовка текста

Далее мы обсудим три программы, которые используются для выделения колонок текста из файлов и их компоновки различными способами.

cut — удаление фрагментов из всех строк в файлах

Программа `cut` используется для извлечения фрагментов текста из строк и вывода их в стандартный вывод. Она может принимать имена файлов в аргументах или данные со стандартного ввода.

Определение фрагментов строк, подлежащих извлечению, реализовано не очень удобно, и для этой цели применяются параметры, перечисленные в табл. 20.3.

Таблица 20.3. Параметры команды cut для выбора фрагментов

Параметр	Длинный параметр	Описание
`-c список_символов`	`--characters= список_символов`	Извлекает фрагмент строки, определяемый `списком_символов`. Список может включать один или несколько числовых диапазонов, разделенных запятыми
`-f список_полей`	`--fields= список_полей`	Извлекает одно или несколько полей из строки, как определено аргументом `список_символов`. Список может включать одно или несколько полей или диапазонов полей, разделенных запятыми
`-d символ_разделитель`	`--delimiter= символ_разделитель`	В присутствии параметра `-f`, в качестве разделителя полей используется `символ_разделитель`. По умолчанию поля должны отделяться друг от друга одним символом табуляции
	`--complement`	Извлекает строку текста целиком, кроме фрагментов, определяемых параметром `-c` и/или `-f`

Как видите, программа `cut` не обладает особенной гибкостью. Она лучше всего подходит для извлечения фрагментов из текста, произведенного другими программами, а не человеком. Давайте вернемся к нашему файлу *distros.txt* и посмотрим, достаточно ли он «хорош» для программы `cut`. Если воспользоваться программой `cat` с параметром `-A`, можно увидеть, отвечает ли файл требованию в отношении использования символа табуляции в качестве разделителя полей.

```
[me@linuxbox ~]$ cat -A distros.txt
SUSE^I10.2^I12/07/2006$
Fedora^I10^I11/25/2008$
SUSE^I11.0^I06/19/2008$
Ubuntu^I8.04^I04/24/2008$
Fedora^I8^I11/08/2007$
SUSE^I10.3^I10/04/2007$
Ubuntu^I6.10^I10/26/2006$
Fedora^I7^I05/31/2007$
Ubuntu^I7.10^I10/18/2007$
Ubuntu^I7.04^I04/19/2007$
SUSE^I10.1^I05/11/2006$
Fedora^I6^I10/24/2006$
Fedora^I9^I05/13/2008$
Ubuntu^I6.06^I06/01/2006$
Ubuntu^I8.10^I10/30/2008$
Fedora^I5^I03/20/2006$
```

Похоже, что все в порядке: пробелы отсутствуют и поля разделены единственными символами табуляции. Поскольку вместо пробелов в файле используются символы табуляции, можно воспользоваться параметром `-f` для извлечения поля:

```
[me@linuxbox ~]$ cut -f 3 distros.txt
12/07/2006
11/25/2008
06/19/2008
04/24/2008
11/08/2007
10/04/2007
10/26/2006
05/31/2007
10/18/2007
04/19/2007
05/11/2006
10/24/2006
05/13/2008
06/01/2006
10/30/2008
03/20/2006
```

Так как поля в файле *distros.txt* разделены символами табуляции, их удобнее извлекать с помощью `cut` именно как поля, а не как группы символов. Когда поля разделяются символами табуляции, маловероятно, что строки будут содержать одно и то же число символов, из-за чего определение позиций символов в строках становится сложной или неразрешимой задачей. В примере, приведенном выше, мы смогли извлечь поля с датами, которые, к нашей удаче, все имеют одинаковую длину, поэтому теперь мы можем показать, как выполняется извлечение групп символов, для чего попробуем извлечь год из каждой строки:

```
[me@linuxbox ~]$ cut -f 3 distros.txt | cut -c 7-10
2006
2008
2008
2008
2007
2007
2006
2007
2007
2007
2006
2006
2008
2006
2008
2006
```

Применив `cut` второй раз к нашему списку, мы смогли извлечь символы в позициях с 7-й по 10-ю, которые соответствуют году в поле с датой. Форма записи `7-10` — это пример определения диапазона. Полное описание особенностей определения диапазонов находится на странице справочного руководства (man) для команды `cut`.

При работе с полями определим разделитель, отличающийся от символа табуляции. Следующий пример извлекает первое поле из файла */etc/passwd*:

```
[me@linuxbox ~]$ cut -d ':' -f 1 /etc/passwd | head
root
daemon
bin
sys
sync
games
man
lp
mail
news
```

С помощью параметра `-d` мы определили, что роль разделителя полей будет играть двоеточие.

ЗАМЕНА СИМВОЛОВ ТАБУЛЯЦИИ

Наш файл *distros.txt* идеально отформатирован для извлечения полей с использованием `cut`. Но что, если нам понадобится обработать файл, вырезая фрагменты по символам, а не по полям? Для этого нам нужно заменить символы табуляции в файле соответствующим числом пробелов. К счастью, в GNU-пакете Coreutils имеется инструмент для этого — программа `expand`. Она может принимать имена файлов в аргументах или данные со стандартного ввода и выводить измененный текст в стандартный вывод.

Если обработать наш файл *distros.txt* программой `expand`, мы сможем использовать `cut -c` для извлечения любых диапазонов символов из файла. Например, с помощью следующей команды можно извлечь год выпуска из нашего файла со списком, применив `cut` для извлечения всех символов с 23-й позиции до конца строки:

```
[me@linuxbox ~]$ expand distros.txt | cut -c 23-
```

В состав пакета Coreutils входит также программа `unexpand`, замещающая пробелы символами табуляции.

paste — слияние строк из файлов

Команда `paste` выполняет операцию, обратную команде `cut`. Вместо извлечения колонок текста из файла она добавляет одну или несколько колонок текста в файл. Для этого она читает содержимое нескольких файлов, комбинирует поля, найденные в них, и выводит результат в стандартный вывод. Подобно программе `cut`, `paste` принимает несколько файлов в аргументах и/или данные со стандартного ввода. Для демонстрации возможностей программы `paste` выполним небольшую хирургическую операцию с файлом *distros.txt*, чтобы получить список выпусков в хронологическом порядке.

Сначала применим команду `sort`, чтобы получить список дистрибутивов, отсортированный по дате выпуска, и сохраним результат в файле *distros-by-date.txt*:

```
[me@linuxbox ~]$ sort -k 3.7nbr -k 3.1nbr -k 3.4nbr distros.txt >
distros-by-date.txt
```

Затем с помощью `cut` извлечем два первых поля (с именами дистрибутивов и номерами версий) и сохраним результат в файле *distro-versions.txt*:

```
[me@linuxbox ~]$ cut -f 1,2 distros-by-date.txt > distros-versions.txt
[me@linuxbox ~]$ head distros-versions.txt
Fedora          10
Ubuntu          8.10
SUSE            11.0
Fedora          9
Ubuntu          8.04
Fedora          8
Ubuntu          7.10
SUSE            10.3
Fedora          7
Ubuntu          7.04
```

Завершая этап подготовки, извлечем даты выпусков и сохраним их в файле *distro-dates.txt*:

```
[me@linuxbox ~]$ cut -f 3 distros-by-date.txt > distros-dates.txt
[me@linuxbox ~]$ head distros-dates.txt
11/25/2008
10/30/2008
06/19/2008
05/13/2008
04/24/2008
11/08/2007
10/18/2007
10/04/2007
05/31/2007
04/19/2007
```

Теперь у нас есть все необходимое. Чтобы завершить процедуру, с помощью `paste` добавим колонку с датами перед названиями и номерами версий дистрибутивов, создав хронологический список. Для этого достаточно просто вызвать `paste` и передать ей файлы в требуемом порядке.

```
[me@linuxbox ~]$ paste distros-dates.txt distros-versions.txt
11/25/2008      Fedora          10
10/30/2008      Ubuntu          8.10
06/19/2008      SUSE            11.0
05/13/2008      Fedora          9
04/24/2008      Ubuntu          8.04
11/08/2007      Fedora          8
10/18/2007      Ubuntu          7.10
10/04/2007      SUSE            10.3
05/31/2007      Fedora          7
04/19/2007      Ubuntu          7.04
12/07/2006      SUSE            10.2
10/26/2006      Ubuntu          6.10
```

```
10/24/2006      Fedora          6
06/01/2006      Ubuntu          6.06
05/11/2006      SUSE            10.1
03/20/2006      Fedora          5
```

join — объединение строк из двух файлов по общему полю

Программа `join` действует подобно `paste`, в том смысле, что добавляет колонки в файл, но делает это по-своему. Операция `join` у многих ассоциируется с *реляционными базами данных*, где она объединяет записи из нескольких таблиц по общему ключевому полю. Программа `join` выполняет ту же операцию. Она объединяет данные из множества файлов, опираясь на общее ключевое поле.

Чтобы понять, как действует операция `join` в реляционной базе данных, представьте очень маленькую базу данных с двумя таблицами, по одной записи в каждой. Первая таблица, с именем CUSTOMERS, имеет три поля: номер клиента (CUSTNUM), имя клиента (FNAME) и фамилия клиента (LNAME):

```
CUSTNUM    FNAME   LNAME
=========  ======  ======
4681934    John    Smith
```

Вторая таблица, с именем ORDERS, имеет четыре поля: номер заказа (ORDERNUM), номер клиента (CUSTNUM), количество (QUAN) и пункт заказа (ITEM):

```
ORDERNUM     CUSTNUM    QUAN   ITEM
==========  =========   =====  ====
3014953305   4681934    1      Blue Widget
```

Обратите внимание, что обе таблицы имеют общее поле CUSTNUM. Это важно, так как оно устанавливает отношение между таблицами.

Применив операцию `join`, мы сможем объединить поля из двух таблиц, чтобы получить желаемый результат, например, для подготовки накладной. Проверяя совпадение значений в полях CUSTNUM обеих таблиц, операция `join` выдаст следующий результат:

```
FNAME   LNAME   QUAN   ITEM
======  ======  =====  ====
John    Smith   1      Blue Widget
```

Для демонстрации программы `join` нам понадобится пара файлов с общим ключом. Возьмем в качестве отправной точки файл *distros-by-date.txt* и из него

сконструируем два дополнительных файла. Первый будет содержать даты выпусков (которые в этом примере будут играть роль общего ключа) и названия дистрибутивов:

```
[me@linuxbox ~]$ cut -f 1,1 distros-by-date.txt > distros-names.txt
[me@linuxbox ~]$ paste distros-dates.txt distros-names.txt > distros-key-names.txt
[me@linuxbox ~]$ head distros-key-names.txt
11/25/2008      Fedora
10/30/2008      Ubuntu
06/19/2008      SUSE
05/13/2008      Fedora
04/24/2008      Ubuntu
11/08/2007      Fedora
10/18/2007      Ubuntu
10/04/2007      SUSE
05/31/2007      Fedora
04/19/2007      Ubuntu
```

И второй — даты выпусков и номера версий:

```
[me@linuxbox ~]$ cut -f 2,2 distros-by-date.txt > distros-vernums.txt
[me@linuxbox ~]$ paste distros-dates.txt distros-vernums.txt >
               distros-keyvernums.txt
[me@linuxbox ~]$ head distros-key-vernums.txt
11/25/2008      10
10/30/2008      8.10
06/19/2008      11.0
05/13/2008      9
04/24/2008      8.04
11/08/2007      8
10/18/2007      7.10
10/04/2007      10.3
05/31/2007      7
04/19/2007      7.04
```

Теперь у нас есть два файла с общим ключом (поле «дата выпуска»). Здесь важно отметить, что файлы должны быть отсортированы по ключевому полю, чтобы программа `join` выдала правильный результат.

```
[me@linuxbox ~]$ join distros-key-names.txt distros-key-vernums.txt | head
11/25/2008 Fedora 10
10/30/2008 Ubuntu 8.10
06/19/2008 SUSE 11.0
05/13/2008 Fedora 9
04/24/2008 Ubuntu 8.04
11/08/2007 Fedora 8
10/18/2007 Ubuntu 7.10
```

```
10/04/2007 SUSE 10.3
05/31/2007 Fedora 7
04/19/2007 Ubuntu 7.04
```

Отметьте также, что по умолчанию в качестве разделителя полей во входных данных `join` использует символы табуляции, а в выводе — пробел. Такое поведение можно изменить с помощью параметров. За дополнительными подробностями обращайтесь к странице справочного руководства (man) для `join`.

Сравнение текста

Довольно часто бывает необходимо сравнить версии текстовых файлов. Для системных администраторов и разработчиков программного обеспечения это особенно важно. Системному администратору, например, может понадобиться сравнить имеющийся конфигурационный файл с предыдущей версией, чтобы понять суть возникшей проблемы. Аналогично, программисту часто бывает необходимо увидеть изменения, происшедшие в программе с течением времени.

comm — построчное сравнение двух сортированных файлов

Программа `comm` сравнивает два текстовых файла, показывая, какие строки в них уникальные, а какие — одинаковые. Для демонстрации создадим с помощью `cat` два почти идентичных файла:

```
[me@linuxbox ~]$ cat > file1.txt
a
b
c
d
[me@linuxbox ~]$ cat > file2.txt
b
c
d
e
```

Затем сравним эти два файла с помощью `comm`:

```
[me@linuxbox ~]$ comm file1.txt file2.txt
a
                b
                c
                d
        e
```

Как видите, `comm` произвела вывод в три колонки. Первая колонка содержит уникальные строки из первого файла, вторая — уникальные строки из второго файла, третья — строки, одинаковые в обоих файлах. Программа `comm` поддерживает параметры в формате `-n`, где `n` может быть числом 1, 2 или 3. При использовании эти параметры определяют номера колонок, вывод которых следует подавить. Например, чтобы вывести только одинаковые строки, нужно подавить вывод колонок 1 и 2:

```
[me@linuxbox ~]$ comm -12 file1.txt file2.txt
b
c
d
```

diff — построчное сравнение файлов

Подобно программе `comm`, `diff` используется для выявления различий между файлами. Однако `diff` намного более сложный инструмент, поддерживающий вывод во множестве форматов и способный обрабатывать сразу огромные коллекции файлов. Программа `diff` часто используется разработчиками программного обеспечения для исследования различий между разными версиями исходного программного кода, потому что позволяет рекурсивно обходить каталоги, которые часто называют *деревьями исходного кода* (source trees). Часто программа diff применяется для создания *diff-файлов*, или *заплат* (patches), которые могут использоваться другими программами, такими как `patch` (о которой рассказывается чуть ниже), для преобразования файлов из одной версии в другую.

Если применить `diff` к файлам из предыдущего примера, можно увидеть стиль вывода результатов ее работы по умолчанию: краткое описание различий между двумя файлами:

```
[me@linuxbox ~]$ diff file1.txt file2.txt
1d0
< a
4a4
> e
```

В формате по умолчанию каждой группе изменений предшествует *команда изменения* (табл. 20.4) в форме *диапазон — операция — диапазон*, описывающая позиции и типы изменений, которые нужно выполнить, чтобы преобразовать первый файл во второй.

В этом формате любой диапазон представлен списком через запятую номеров начальной и конечной строки. Хотя этот формат используется по умолчанию

(главным образом для совместимости со стандартом POSIX и обратной совместимости с традиционными версиями `diff` для Unix), он не так широко используется, как другие, дополнительные форматы. Два других формата, получивших большую популярность, — это *контекстный формат* и *унифицированный формат*.

Таблица 20.4. Команды изменения, генерируемые программой diff

Команда	Описание
`r1ar2`	Добавить строки, находящиеся в диапазоне `r2` во втором файле, после строк в позиции `r1` в первом файле
`r1cr2`	Изменить (заменить) строки в диапазоне `r1` в первом файле строками в диапазоне `r2` во втором файле
`r1dr2`	Удалить строки в диапазоне `r1` в первом файле, которые находились бы в диапазоне `r2` во втором файле

При использовании контекстного формата (параметр `-c`) вывод выглядит так:

```
[me@linuxbox ~]$ diff -c file1.txt file2.txt
*** file1.txt    2008-12-23 06:40:13.000000000 -0500
--- file2.txt    2008-12-23 06:40:34.000000000 -0500
***************
*** 1,4 ****
- a
  b
  c
  d
--- 1,4 ----
  b

  c
  d
+ e
```

Вывод начинается с имен двух файлов и времени последнего их изменения. Первый файл отмечается звездочками, а второй — дефисами. На протяжении всей оставшейся части листинга эти маркеры обозначают соответствующие им файлы. Далее следуют группы изменений, включая заданное по умолчанию число окружающих строк, определяющих контекст. Первая группа начинается со строки:

```
*** 1,4 ****
```

указывающей на строки с номерами с 1-го по 4-й в первом файле. Далее следует строка:

```
--- 1,4 ----
```

указывающая на строки с номерами с 1-го по 4-й во втором файле. Внутри группы изменений присутствуют строки, начинающиеся с одного из четырех индикаторов, перечисленных в табл. 20.5.

Таблица 20.5. Индикаторы изменений, генерируемые программой diff при использовании контекстного формата

Индикатор	Значение
(нет)	Строка показана для контекста. В ней отсутствуют различия между файлами
-	Строка удалена. Эта строка присутствует в первом файле и отсутствует во втором
+	Строка добавлена. Эта строка присутствует во втором файле и отсутствует в первом
!	Строка изменена. Выводятся две версии строки, каждая в соответствующем разделе внутри группы изменений

Унифицированный формат напоминает контекстный, но более компактный. Задается параметром -u:

```
[me@linuxbox ~]$ diff -u file1.txt file2.txt
--- file1.txt    2008-12-23 06:40:13.000000000 -0500
+++ file2.txt    2008-12-23 06:40:34.000000000 -0500
@@ -1,4 +1,4 @@
-a
 b
 c
 d
+e
```

Самое большое отличие между контекстным и унифицированным форматами — отсутствие повторяющихся контекстных строк, благодаря чему обеспечивается большая компактность унифицированного формата в сравнении с контекстным. В примере, приведенном выше, видны те же времена последнего изменения файлов, что и в контекстном формате, за которыми следует строка `@@ -1,4 +1,4 @@`. Она указывает номера строк в первом и во втором файлах, описываемых группой

изменений. Далее следуют сами строки с тремя (по умолчанию) строками контекста. Каждая строка начинается с одного из трех возможных символов, значение которых описывается в табл. 20.6.

Таблица 20.6. Индикаторы изменений, генерируемые программой diff при использовании унифицированного формата

Индикатор	Значение
(нет)	Строка, одинаковая в обоих файлах
-	Строка отсутствует в первом файле
+	Строка присутствует только в первом файле

patch — применение diff-файла к оригиналу

Программа `patch` используется для применения изменений к текстовым файлам. Она принимает вывод программы `diff` и обычно используется для преобразования старых версий файлов в более новые. Рассмотрим один известный пример. Ядро Linux разрабатывается большой, свободно организованной группой разработчиков, от которых неиссякаемым потоком идут небольшие изменения в исходном коде. Ядро Linux включает миллионы строк программного кода, но изменения, присылаемые одним разработчиком за один раз, очень невелики. Разработчикам нет смысла пересылать все дерево исходных текстов ядра всякий раз, когда вносится небольшое изменение. Вместо этого они присылают diff-файлы. Эти файлы описывают различия между предыдущей версией ядра и новой, включающей изменения разработчика. Другой разработчик, получивший такое изменение, использует программу `patch`, позволяющую применить предложенное изменение к своему дереву исходных текстов. Использование пары программ `diff`/`patch` дает два важных преимущества:

- diff-файл очень невелик в сравнении с полным деревом исходных текстов;
- diff-файл наглядно показывает произведенные изменения, что позволяет экспертам быстро применить эти изменения и оценить их.

Разумеется, пару `diff`/`patch` можно применять к любым текстовым файлам, не только к исходному коду. Эти программы с таким же успехом можно применять к конфигурационным файлам или другому тексту.

Чтобы подготовить diff-файл для последующего его применения программой `patch`, документация GNU предлагает использовать `diff`, как показано ниже:

```
diff -Naur старый_файл новый_файл > diff_файл
```

где `старый_файл` и `новый_файл` могут быть одиночными файлами или каталогами. Параметр `r` поддерживает рекурсивный обход вложенных подкаталогов.

Получив diff-файл, его можно применить к старому файлу, чтобы получить новый:

```
patch < diff_файл
```

Продемонстрируем это на примере нашего тестового файла:

```
[me@linuxbox ~]$ diff -Naur file1.txt file2.txt > patchfile.txt
[me@linuxbox ~]$ patch < patchfile.txt
patching file file1.txt
[me@linuxbox ~]$ cat file1.txt
b
c
d
e
```

В этом примере мы создали diff-файл с именем *patchfile.txt* и затем воспользовались программой `patch`, чтобы применить его (наложить «заплату»). Обратите внимание, что нам не пришлось указывать целевой файл, потому что diff-файл (в унифицированном формате) уже содержит имена файлов в заголовке. После наложения «заплаты» содержимое *file1.txt* точно соответствует содержимому *file2.txt*.

Программа `patch` имеет большое число параметров, а кроме того, существует множество вспомогательных программ, которые помогут в правке «заплат» (diff-файлов).

Редактирование на лету

Наш опыт работы с текстовыми редакторами ограничивается в основном интерактивным способом их использования, в том смысле, что мы вручную перемещаем курсор в нужное место и затем вносим необходимые изменения. Однако существуют также *неинтерактивные* способы редактирования текста. Вполне возможно, например, применить серию изменений к множеству файлов всего одной командой.

tr — перекодирование или удаление символов

Программа `tr` используется для *перекодирования* символов. Ее можно рассматривать как своеобразную посимвольную операцию поиска с заменой. Переко-

дирование — это процесс замены символов из одного алфавита символами из другого алфавита. Например, преобразование символов из нижнего регистра в верхний — это перекодирование. Такое преобразование можно выполнить с помощью `tr`:

```
[me@linuxbox ~]$ echo "lowercase letters" | tr a-z A-Z
LOWERCASE LETTERS
```

Как видите, `tr` принимает исходные данные со стандартного ввода и выводит результаты в стандартный вывод. `tr` принимает два аргумента: множество символов, подлежащих преобразованию, и соответствующее множество символов, в которые должны превратиться преобразуемые символы. Множества символов можно выразить тремя способами:

- в виде списка-перечисления, например `ABCDEFGHIJKLMNOPQRSTUVWXYZ`;
- в виде диапазона символов, например `A-Z`. Обратите внимание, что этому способу сопутствуют те же проблемы, что наблюдаются в других программах (из-за разного порядка алфавитной сортировки в разных регионах), и потому он должен использоваться с осторожностью;
- в виде классов символов POSIX, например `[:upper:]`.

В большинстве случаев множества символов должны иметь одинаковую длину; однако вполне допустимо, если первое множество окажется больше второго. Это может пригодиться, например, если потребуется преобразовать несколько символов в один:

```
[me@linuxbox ~]$ echo "lowercase letters" | tr [:lower:] A
AAAAAAAAA AAAAAAA
```

Кроме перекодирования `tr` позволяет просто удалять символы из входного потока. Выше в этой главе мы обсуждали проблему преобразования текстовых файлов в формате MS-DOS в текст в формате Unix. Для такого преобразования достаточно просто удалить символы возврата каретки в конце каждой строки. Эту операцию можно выполнить с помощью `tr`:

```
tr -d '\r' < файл_dos > файл_unix
```

где `файл_dos` — это файл, подлежащий преобразованию, а `файл_unix` — результат. В этой форме команды используется экранированная последовательность `\r`, представляющая символ возврата каретки. Чтобы увидеть полный список последовательностей и классов символов, поддерживаемых программой `tr`, попробуйте:

```
[me@linuxbox ~]$ tr --help
```

`tr` также позволяет выполнять и другие трюки. При вызове с параметром `-s` команда `tr` «сжимает» (squeeze), или удаляет, повторяющиеся экземпляры символов:

```
[me@linuxbox ~]$ echo "aaabbbccc" | tr -s ab
abccc
```

Здесь у нас имеется строка с повторяющимися символами. Передав множество `ab` команде `tr`, мы удалили повторяющиеся экземпляры символов, входящие в множество, при этом остальные символы (`c`), отсутствующие в множестве, остались нетронутыми. Обратите внимание, что повторяющиеся символы должны следовать подряд. В противном случае сжатие не даст никакого эффекта:

```
[me@linuxbox ~]$ echo "abcabcabc" | tr -s ab
abcabcabc
```

ROT13: НЕ САМЫЙ НАДЕЖНЫЙ СПОСОБ ШИФРОВАНИЯ

Одно забавное применение команды `tr` — шифрование текста по алгоритму ROT13. ROT13 — тривиальный тип шифрования, основанный на простом подстановочном шифре. *Шифрованием* назвать этот алгоритм можно только с большой натяжкой, скорее это алгоритм обфускации (запутывания) текста. Он используется иногда для запутывания потенциально уязвимого содержимого. Метод заключается в простом смещении каждого символа на 13 позиций далее по алфавиту. Так как число 13 соответствует середине набора из 26 символов, повторное применение алгоритма к тексту приводит к его восстановлению в исходное состояние. Шифрование с помощью `tr` выполняется, как показано ниже:

```
echo "secret text" | tr a-zA-Z n-za-mN-ZA-M
frperg grkg
```

Повторное применение той же процедуры приводит к обратному преобразованию:

```
echo "frperg grkg" | tr a-zA-Z n-za-mN-ZA-M
secret text
```

Многие программы для работы с электронной почтой и чтения новостей Usenet поддерживают шифрование ROT13. В Википедии можно найти замечательную статью по этой теме: http://ru.wikipedia.org/wiki/ROT13.

sed — потоковый редактор для фильтрации и преобразования текста

Имя `sed` — это сокращенное словосочетание *stream editor* (потоковый редактор). Данная команда осуществляет редактирование потока текста, получаемого из множества файлов или подаваемого на стандартный ввод команды. `sed` — мощная и достаточно сложная программа (ей посвящены целые книги), поэтому здесь мы не будем рассматривать ее во всех подробностях.

В общем случае `sed` используется следующим образом: ей передается единственная команда редактирования (в командной строке) или имя файла сценария с множеством команд, и она применяет эти команды к каждой строке в потоке текста. Ниже приводится очень простой пример `sed` в действии:

```
[me@linuxbox ~]$ echo "front" | sed 's/front/back/'
back
```

В этом примере с помощью `echo` создается поток текста с единственным словом, который по конвейеру передается программе `sed`. `sed`, в свою очередь, применяет инструкцию `s/front/back/` к тексту в потоке и выводит результат. Эта команда напоминает команду подстановки (поиск с заменой) в редакторе `vi`.

Команды `sed` начинаются с единственной буквы. В примере, рассмотренном выше, буква `s` представляет команду подстановки (substitution). За ней следуют искомая строка и строка замены, разделенные слешем. В качестве разделителя можно использовать любые символы. По общепринятому соглашению, чаще других используется символ «слеш», но `sed` будет использовать в качестве разделителя любой символ, следующий сразу за командой. Ту же самую команду можно было бы записать иначе:

```
[me@linuxbox ~]$ echo "front" | sed 's_front_back_'
back
```

Символ подчеркивания, следующий сразу за командой, становится разделителем. Возможность употребления произвольных разделителей можно использовать для улучшения читаемости команд, как будет показано далее.

Большинству команд в `sed` может предшествовать адрес, который определяет, какие строки во входном потоке должны редактироваться. Если адрес отсутствует, команда редактирования применяется ко всем строкам во входном потоке.

В простейшем случае адрес — это номер строки. Мы могли бы добавить единицу в наш пример:

```
[me@linuxbox ~]$ echo "front" | sed '1s/front/back/'
back
```

Добавление адреса `1` в команду гарантирует применение операции подстановки только к первой строке в нашем однострочном потоке. Можно указать другое число:

```
[me@linuxbox ~]$ echo "front" | sed '2s/front/back/'
front
```

Теперь, как видите, редактирование не было выполнено, потому что во входном потоке отсутствует строка с номером 2.

Адреса можно выражать множеством способов. В табл. 20.7 перечислены адреса, чаще других используемые на практике.

Таблица 20.7. Форма записи адресов в команде sed

Адрес	Описание
`n`	Номер строки, где `n` — положительное число
`$`	Последняя строка
`/регулярное_выражение/`	Строки, соответствующие простому регулярному выражению POSIX. Обратите внимание, что регулярное выражение должно ограничиваться символ «слеш» с обеих сторон. При желании можно использовать другие ограничительные символы, определив регулярное выражение в форме `\cрегулярное_выражениеc`, где `c` — альтернативный символ-ограничитель
`адр1,адр2`	Диапазон строк с номерами от `адр1` по `адр2` включительно. Каждый адрес может иметь любую форму из перечисленных выше
`первая~шаг`	Соответствует строке с номером `первая` и каждой последующей с указанным `шагом`. Например, адрес `1~2` соответствует всем строкам с нечетными номерами, а адрес `5~5` соответствует пятой строке и каждой пятой последующей
`адр1,+n`	Соответствует строке с адресом `адр1` и следующим за ней `n` строкам
`адр!`	Соответствует всем строкам, кроме строки с адресом `адр`, где адрес может иметь любую форму из перечисленных выше

Рассмотрим разные способы адресации строк на примере файла *distros.txt*, созданного выше в этой главе. Сначала попробуем диапазоны номеров строк:

```
[me@linuxbox ~]$ sed -n '1,5p' distros.txt
SUSE            10.2    12/07/2006
Fedora          10      11/25/2008
SUSE            11.0    06/19/2008
Ubuntu          8.04    04/24/2008
Fedora          8       11/08/2007
```

В нашем примере мы вывели строки с 1-й по 5-ю. Для этого использовалась команда `p`, которая просто выводит строки, соответствующие адресам. Однако здесь нам пришлось добавить параметр `-n` (параметр подавления автоматического вывода), чтобы программа `sed` не выводила все строки, что она делает по умолчанию.

Далее попробуем задействовать регулярное выражение:

```
[me@linuxbox ~]$ sed -n '/SUSE/p' distros.txt
SUSE            10.2    12/07/2006
SUSE            11.0    06/19/2008
SUSE            10.3    10/04/2007
SUSE            10.1    05/11/2006
```

Добавив регулярное выражение `/SUSE/`, заключенное в слеши, мы смогли выделить строки подобно тому, как это делает программа `grep`.

Наконец, попробуем применить оператор отрицания, добавив в адрес восклицательный знак (`!`):

```
[me@linuxbox ~]$ sed -n '/SUSE/!p' distros.txt
Fedora          10      11/25/2008
Ubuntu          8.04    04/24/2008
Fedora          8       11/08/2007
Ubuntu          6.10    10/26/2006
Fedora          7       05/31/2007
Ubuntu          7.10    10/18/2007
Ubuntu          7.04    04/19/2007
Fedora          6       10/24/2006
Fedora          9       05/13/2008
Ubuntu          6.06    06/01/2006
Ubuntu          8.10    10/30/2008
Fedora          5       03/20/2006
```

Здесь мы видим ожидаемый результат: все строки из файла, кроме совпавших с регулярным выражением.

Пока что мы познакомились лишь с двумя командами редактирования, поддерживаемыми программой `sed`: `s` и `p`. В табл. 20.8 приводится более полный список основных команд редактирования.

Таблица 20.8. Основные команды редактирования sed

Команда	Описание
`=`	Выводит номер текущей строки
`a`	Добавляет текст в конец текущей строки
`d`	Удаляет текущую строку
`i`	Вставляет текст в начало текущей строки
`p`	Выводит текущую строку. По умолчанию `sed` выводит все строки, но редактирует только соответствующие указанному адресу. Поведение по умолчанию можно отменить, передав параметр `-n`
`q`	Завершает `sed` без обработки остальных строк. Если параметр `-n` не указан, выводит текущую строку
`Q`	Завершает `sed` без обработки остальных строк
`s/регулярное_выражение/строка_замены/`	Замещает совпадение с регулярным выражением строкой замены. Строка замены может включать специальный символ &, обозначающий совпадение с регулярным выражением. Кроме того, строка замены может включать последовательности, с `\1` по `\9`, обозначающие совпадения с соответствующими подвыражениями в регулярном выражении. Дополнительную информацию по этой теме можно найти в обзоре обратных ссылок ниже. За символом «слеш», закрывающим строку замены, может следовать необязательный флаг, определяющий дополнительные особенности поведения команды
`y/множество1/множество2`	Выполняет перекодирование, преобразуя символы из первого множества в символы второго множества. Имейте в виду, что в отличие от программы `tr`, `sed` требует, чтобы оба множества были одинаковой длины

Команда `s`, вне всяких сомнений, используется намного чаще других команд редактирования. Далее мы рассмотрим только часть ее возможностей, выполняя редактирование нашего файла *distros.txt*. Мы уже говорили, что поле даты в *distros.txt* хранит информацию не в самом «дружественном» для компьютеров виде. Здесь даты записаны в формате ММ/ДД/ГГГГ, однако гораздо удобнее (для сортировки) было бы, если бы даты были записаны в формате ГГГГ-ММ-ДД. Замена

представления дат вручную — довольно утомительное занятие и чревато ошибками, но с помощью `sed` ту же замену можно выполнить в одно действие:

```
[me@linuxbox ~]$ sed 's/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\
                    2/' distros.txt
SUSE             10.2    2006-12-07
Fedora           10      2008-11-25
SUSE             11.0    2008-06-19
Ubuntu           8.04    2008-04-24
Fedora           8       2007-11-08
SUSE             10.3    2007-10-04
Ubuntu           6.10    2006-10-26
Fedora           7       2007-05-31
Ubuntu           7.10    2007-10-18
Ubuntu           7.04    2007-04-19
SUSE             10.1    2006-05-11
Fedora           6       2006-10-24
Fedora           9       2008-05-13
Ubuntu           6.06    2006-06-01
Ubuntu           8.10    2008-10-30
Fedora           5       2006-03-20
```

Прекрасный результат! Правда, команда выглядит устрашающе, но она работает. За один шаг мы изменили представление дат во всем файле. Этот пример также наглядно показывает, почему про регулярные выражения иногда в шутку говорят «только для записи». Мы можем писать их, но прочитать их порой никак не получается. Прежде чем сбежать от этой устрашающей команды, давайте посмотрим, как она была сконструирована. Во-первых, как мы уже знаем, эта команда имеет следующую структуру:

```
sed 's/регулярное_выражение/строка_замены/' distros.txt
```

Теперь разберем регулярное выражение, отыскивающее даты. Так как даты имеют формат ММ/ДД/ГГГГ и находятся в конце строки, найти их можно с помощью следующего выражения:

```
[0-9]{2}/[0-9]{2}/[0-9]{4}$
```

которому соответствуют две цифры, слеш, две цифры, слеш, четыре цифры и конец строки. Так, с регулярным выражением разобрались, а что со строкой замены? Чтобы описать ее, нам необходимо познакомиться с новой для нас особенностью регулярных выражений, которую можно использовать в некоторых приложениях, поддерживающих BRE. Эта особенность называется *обратные ссылки*, и действует она так: если в строке замены присутствует последовательность `\n`, где `n` — число от одного до девяти, эта последовательность будет ссылаться на совпадение

с соответствующим подвыражением в предшествующем регулярном выражении. Чтобы создать подвыражение, достаточно просто заключить часть регулярного выражения в круглые скобки, например:

```
([0-9]{2})/([0-9]{2})/([0-9]{4})$
```

Теперь у нас есть три подвыражения. Первому соответствует месяц, второму — число месяца и третьему — год. Соответственно строку замены можно выразить так:

```
\3-\1-\2
```

что даст нам в результате такую последовательность: год, дефис, месяц, дефис, число месяца.

Теперь наша команда приобрела следующий вид:

```
sed 's/([0-9]{2})/([0-9]{2})/([0-9]{4})$/\3-\1-\2/' distros.txt
```

Но остались две проблемы. Первая: дополнительные слеши в регулярном выражении запутают программу `sed`, когда она попытается интерпретировать команду `s`. Вторая: так как по умолчанию `sed` принимает только простые регулярные выражения, некоторые символы в нашем регулярном выражении будут интерпретироваться как литералы, а не как метасимволы. Мы решим обе проблемы, применив обратные слеши для экранирования нужных символов:

```
sed 's/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/' distros.txt
```

И дело в шляпе!

Другая особенность команды `s` — возможность использования дополнительных флагов вслед за строкой замены. Наиболее примечательным из них является флаг `g`, который требует от `sed` применить поиск с заменой к строке глобально (globally), а не только к первому найденному совпадению, как это делается по умолчанию. Например:

```
[me@linuxbox ~]$ echo "aaabbbccc" | sed 's/b/B/'
aaaBbbccc
```

Как видите, замена была выполнена только для первого вхождения буквы *b*, а остальные остались нетронутыми. Добавив флаг `g`, можно изменить все вхождения:

```
[me@linuxbox ~]$ echo "aaabbbccc" | sed 's/b/B/g'
aaaBBBccc
```

До сих пор мы передавали команды программе `sed` только по одной и только в командной строке. Однако существует возможность создавать более сложные команды в файлах сценариев и передавать эти сценарии с помощью параметра `-f`. Для демонстрации создадим с помощью `sed` отчет на основе нашего файла *distros.txt*. Отчет будет содержать заголовок вверху, измененные даты и названия дистрибутивов будут преобразованы в верхний регистр. Для этого нам понадобится написать сценарий, поэтому запустите текстовый редактор и введите следующие строки:

```
# Сценарий для sed, создающий отчет о дистрибутивах Linux

1 i\
\
Linux Distributions Report\
s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

Сохраните сценарий в файл с именем *distros.sed* и запустите его:

```
[me@linuxbox ~]$ sed -f distros.sed distros.txt

Linux Distributions Report

SUSE            10.2    2006-12-07
FEDORA          10      2008-11-25
SUSE            11.0    2008-06-19
UBUNTU          8.04    2008-04-24
FEDORA          8       2007-11-08
SUSE            10.3    2007-10-04
UBUNTU          6.10    2006-10-26
FEDORA          7       2007-05-31
UBUNTU          7.10    2007-10-18
UBUNTU          7.04    2007-04-19
SUSE            10.1    2006-05-11
FEDORA          6       2006-10-24
FEDORA          9       2008-05-13
UBUNTU          6.06    2006-06-01
UBUNTU          8.10    2008-10-30
FEDORA          5       2006-03-20
```

Как видите, сценарий выдал желаемый результат, но как он это сделал? Давайте вернемся еще раз к нашему сценарию. Выведем его с помощью программы `cat` так, чтобы она пронумеровала строки:

```
[me@linuxbox ~]$ cat -n distros.sed
   1 # Сценарий для sed, создающий отчет о дистрибутивах Linux
```

```
2
3 1 i\
4 \
5 Linux Distributions Report\
6
7 s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
8 y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

Строка с номером 1 — это *комментарий*. Так же как во многих конфигурационных файлах и языках программирования, широко используемых в Linux, комментарии начинаются с символа `#`, за которым следует пояснительный текст. Комментарии можно помещать в сценарии в любое место (только не в сами команды), и они, безусловно, полезны для всех, кто хочет выяснить особенности работы сценария или сопровождать его.

Строка 2 — это пустая строка. Так же как комментарии, мы можем добавлять пустые строки для удобочитаемости.

Многие команды `sed` поддерживают адресацию строк. Адреса используются, чтобы определить, к каким строкам во входном потоке применяется операция. Адреса могут выражаться в форме простых номеров строк, диапазонов номеров и специального номера `$`, соответствующего последней входной строке.

В строках с 3-й по 6-ю содержится текст, который должен быть вставлен по адресу 1, в первую входную строку. Команда `i`, сопровождаемая последовательностью из обратного слеша и перевода строки, производит экранированный символ перевода, или то, что мы называем *символом продолжения строки*. Эта последовательность используется во многих случаях, в том числе и в сценариях на языке командной оболочки, позволяя встраивать символ перевода строки в поток текста так, чтобы он не воспринимался интерпретатором (в данном случае программой `sed`) как конец строки. Команда `i`, а также команда `a` (добавления текста в конец) и команда `c` (замены текста) могут располагаться в нескольких строках текста, при условии, что каждая из них, кроме последней, будет завершаться символом продолжения строки. Шестая строка в нашем сценарии фактически завершает вставляемый текст и заканчивается уже не символом продолжения строки, а простым переводом строки, сообщая о завершении команды `i`.

ПРИМЕЧАНИЕ

Символ продолжения строки формируется из обратного слеша и следующего сразу за ним перевода строки. Никаких промежуточных пробелов между ними быть не должно.

Строка 7 — наша команда поиска с заменой. Так как ей не предшествует никакой конкретный адрес, она будет выполнена для каждой строки во входном потоке.

Строка 8 выполняет перекодировку букв нижнего регистра в буквы верхнего регистра. Обратите внимание, что, в отличие от программы `tr`, команда `y` в `sed` не поддерживает ни диапазоны символов (например, `[a-z]`), ни классы символов POSIX. И снова, так как команде `y` не предшествует никакой конкретный адрес, она будет выполнена для каждой строки во входном потоке.

ИСПОЛЬЗУЮЩИЕ SED ТАКЖЕ ЧАСТО ВЫБИРАЮТ...

Программа `sed` обладает очень широкими возможностями. С ее помощью можно решать весьма сложные задачи, связанные с редактированием потока текста. Но чаще она используется для выполнения простеньких операций, определение которых укладывается в одну строку. Для решения объемных задач многие предпочитают использовать другие инструменты. Наиболее популярными из них являются `awk` и `perl`. Они не относятся к разряду простых инструментов, как программы, рассматриваемые здесь, а являются полноценными языками программирования. `perl`, например, часто применяется взамен языка командной оболочки для решения многих задач системного администрирования, а также пользуется большой популярностью как средство разработки веб-приложений. `awk` имеет более узкую область применения. Основное его достоинство заключается в возможности управления табличными данными. Он напоминает `sed` в том смысле, что программы на `awk` обычно занимаются построчной обработкой текстовых файлов, используя схему, похожую на адреса в `sed` со следующими за ними операциями. Даже при том, что рассмотрение `awk` и `perl` выходит за рамки этой книги, они являются отличными инструментами для пользователей командной строки в Linux.

aspell — интерактивная проверка орфографии

Последний инструмент, который мы рассмотрим в этой главе, — программа `aspell`, интерактивное средство проверки орфографии. Программа `aspell` является преемницей программы `ispell`, существовавшей прежде, и может использоваться как ее замена. Чаще всего программа `aspell` используется другими программами в тех случаях, когда необходима функция проверки орфографии, однако `aspell` может также весьма эффективно использоваться как самостоятельный инструмент командной строки. Она способна проверять текстовые файлы разных типов, включая документы HTML, программы на C/C++, электронные письма и другие специальные виды текста.

Чтобы проверить орфографию в файле с простым текстом, можно вызвать `aspell`, как показано ниже:

```
aspell check текстовый_файл
```

где *`текстовый_файл`* — это имя файла для проверки. В качестве практического примера создадим простой текстовый файл с именем *foo.txt*, содержащий несколько произвольных орфографических ошибок:

```
[me@linuxbox ~]$ cat > foo.txt
The quick brown fox jimped over the laxy dog.
```

Затем проверим файл с помощью `aspell`:

```
[me@linuxbox ~]$ aspell check foo.txt
```

Поскольку в режиме проверки (check) программа `aspell` действует интерактивно, вы увидите следующий экран:

```
The quick brown fox jimped over the laxy dog.

1) jumped                                6) wimped
2) gimped                                7) camped
3) comped                                8) humped
4) limped                                9) impede
5) pimped                                0) umped
i) Ignore                                I) Ignore all
r) Replace                               R) Replace all
a) Add                                   l) Add Lower
b) Abort                                 x) Exit

?
```

В верхней части экрана выводится текст с выделенным подозрительным словом. В середине — 10 вариантов исправления ошибки, пронумерованных от 0 до 9, а затем следует список других возможных действий. Наконец, в самом низу выводится приглашение к вводу, готовое принять наш выбор.

Если ввести `1`, `aspell` заменит подозрительное слово *jimped* словом *jumped* и перейдет к следующему подозрительному слову, *laxy*. Если выбрать вариант замены *lazy*, `aspell` выполнит подстановку и завершится (так как ошибок во введенной фразе больше нет). После того как `aspell` завершится, заглянем в файл и увидим, что все ошибки исправлены:

```
[me@linuxbox ~]$ cat foo.txt
The quick brown fox jumped over the lazy dog.
```

Если вызвать программу `aspell` без параметра `--dont-backup`, она создаст резервную копию файла с исходным текстом, добавив к имени файла расширение *.bak*.

А теперь похвастаемся умением пользоваться программой `sed` и вернем наши ошибки на место, чтобы продолжить эксперименты с нашим файлом:

```
[me@linuxbox ~]$ sed -i 's/lazy/laxy/; s/jumped/jimped/' foo.txt
```

Параметр `-i` сообщает программе `sed`, что требуется отредактировать файл «на месте», в том смысле, что изменения нужно произвести в самом файле, а не переслать их в стандартный вывод. Здесь также показана возможность передать более одной команды редактирования, разделив их точкой с запятой.

Далее мы посмотрим, как `aspell` справляется с текстовыми файлами разных видов. С помощью текстового редактора, например `vim` (наиболее смелые могут попробовать использовать `sed`), добавим в файл немного разметки HTML:

```
<html>
    <head>
        <title>Mispelled HTML file</title>
    </head>
    <body>
        <p>The quick brown fox jimped over the laxy dog.</p>
    </body>
</html>
```

Если теперь попытаться проверить орфографию в измененном файле, мы столкнемся с проблемой. Вызвав команду

```
[me@linuxbox ~]$ aspell check foo.txt
```

мы получим следующее:

```
<html>
    <head>
        <title>Mispelled HTML file</title>
    </head>
    <body>
        <p>The quick brown fox jimped over the laxy dog.</p>
    </body>
</html>

1) HTML                                    4) Hamel
2) ht ml                                   5) Hamil
3) ht-ml                                   6) hotel
i) Ignore                                  I) Ignore all
r) Replace                                 R) Replace all
```

```
a) Add                                  l) Add Lower
b) Abort                                x) Exit

?
```

`aspell` посчитала, что HTML-теги записаны с ошибками. Эту проблему можно преодолеть, передав параметр `-H` (HTML) режима проверки:

```
[me@linuxbox ~]$ aspell -H check foo.txt
```

Теперь результат будет выглядеть так:

```
<html>
    <head>
        <title>Mispelled HTML file</title>
    </head>
    <body>
        <p>The quick brown fox jimped over the laxy dog.</p>
    </body>
</html>

1) Mi spelled                           6) Misapplied
2) Mi-spelled                           7) Miscalled
3) Misspelled                           8) Respelled
4) Dispelled                            9) Misspell
5) Spelled                              0) Misled
i) Ignore                               I) Ignore all
r) Replace                              R) Replace all
a) Add                                  l) Add Lower
b) Abort                                x) Exit

?
```

Теперь теги HTML игнорируются, и проверке подвергаются только фрагменты файла, не являющиеся частью разметки. В этом режиме содержимое HTML-тегов игнорируется и не проверяется, исключение составляет содержимое тегов `ALT` (точнее, атрибутов `alt`), которое будет проверяться в этом режиме проверки.

ПРИМЕЧАНИЕ

По умолчанию `aspell` игнорирует адреса URL и электронной почты в тексте. Эту ситуацию можно изменить с помощью параметров командной строки. Также можно указать, какие теги разметки должны проверяться, а какие пропускаться. За подробностями обращайтесь к странице справочного руководства (man) для `aspell`.

Заключение

В этой главе мы познакомились с несколькими из множества инструментов командной строки для обработки текста. В следующей главе мы рассмотрим еще несколько. Нужно признать, что для многих из вас пока не очевидно, как или для чего можно было бы использовать некоторые из них в повседневной работе, хотя мы попытались привести практические примеры. В следующих главах вы увидите, что эти инструменты формируют базовый набор для решения большого количества практических задач. Это вам пригодится, когда мы перейдем к сценариям на языке командной оболочки, где эти инструменты по-настоящему продемонстрируют свои возможности.

Дополнительное задание

Существует несколько интересных команд обработки текста, на которые стоит обратить внимание. Среди них `split` (разбивает файлы на фрагменты), `csplit` (разбивает файлы на фрагменты, опираясь на контекст) и `sdiff` (выводит различия между файлами, что называется, «бок о бок»).

21 Форматирование вывода

В этой главе мы продолжим знакомство с инструментами, имеющими отношение к тексту, сосредоточившись на программах для форматирования выводимого текста, а не его изменения. Эти инструменты часто используются для подготовки текста к печати, о которой мы поговорим в следующей главе. В этой главе мы рассмотрим следующие программы:

`nl` — нумерует строки;

`fold` — выполняет перенос строк, ограничивая их указанной длиной;

`fmt` — выполняет простое форматирование текста;

`pr` — форматирует текст для печати;

`printf` — форматирует и выводит данные;

`groff` — система форматирования документов.

Инструменты простого форматирования

Для начала рассмотрим несколько инструментов простого форматирования. В основном это узкоспециализированные и довольно бесхитростные программы, но их можно использовать для решения простых задач в конвейерах и сценариях.

nl — нумерация строк

Программа `nl` предназначена для решения простой задачи: она выполняет нумерацию строк. В простейшем случае использования `nl` напоминает команду `cat -n`:

```
[me@linuxbox ~]$ nl distros.txt | head
     1	SUSE	10.2	12/07/2006
     2	Fedora	10	11/25/2008
     3	SUSE	11.0	06/19/2008
     4	Ubuntu	8.04	04/24/2008
     5	Fedora	8	11/08/2007
     6	SUSE	10.3	10/04/2007
     7	Ubuntu	6.10	10/26/2006
     8	Fedora	7	05/31/2007
     9	Ubuntu	7.10	10/18/2007
    10	Ubuntu	7.04	04/19/2007
```

Так же, как `cat`, программа `nl` может принимать несколько имен файлов в аргументах командной строки или данные со стандартного ввода. Однако `nl` имеет ряд параметров и поддерживает простейшую форму разметки, обеспечивая более сложные способы нумерации.

`nl` поддерживает идею *логических страниц*. Это дает возможность начинать нумерацию на каждой странице заново. С помощью параметров можно определить номер первой строки и протяженность нумерации, а также формат номеров. Логическую страницу можно разбить на заголовок, тело и нижний колонтитул. В каждом разделе нумерация может начинаться с начала и/или может использоваться разный формат нумерации. Если программе `nl` передать несколько файлов, она будет интерпретировать их как один поток текста. Разделы в потоке выделяются добавлением в поток немного странной разметки, как показано в табл. 21.1.

Таблица 21.1. Разметка nl

Разметка	Значение
`\:\:\:`	Начало заголовка логической страницы
`\:\:`	Начало тела логической страницы
`\:`	Начало нижнего колонтитула логической страницы

Каждый элемент разметки из перечисленных в табл. 21.1 должен находиться в отдельной строке. После обработки элемента программа `nl` удалит его из потока текста.

В табл. 21.2 перечислены наиболее часто используемые параметры `nl`.

Таблица 21.2. Наиболее часто используемые параметры nl

Параметр	Значение
`-b стиль`	Стиль нумерации тела, где аргумент `стиль` может иметь следующие значения: • `a` — нумеровать все строки; • `t` — нумеровать только непустые строки. Этот стиль применяется по умолчанию; • `n` — не нумеровать; • *регулярное выражение* — нумеровать только строки, соответствующие простому регулярному выражению
`-f стиль`	Стиль нумерации нижнего колонтитула. По умолчанию имеет значение `n` (нет нумерации)
`-h стиль`	Стиль нумерации заголовка. По умолчанию имеет значение `n` (нет нумерации)
`-i число`	Шаг приращения номеров на странице. По умолчанию имеет значение 1
`-n формат`	Формат номеров, где аргумент `формат` может иметь следующие значения: • `ln` — с выравниванием по левому краю, без ведущих нулей; • `rn` — с выравниванием по правому краю, без ведущих нулей. Используется по умолчанию; • `rz` — с выравниванием по правому краю и с ведущими нулями
`-p`	Не сбрасывать нумерацию в начале каждой логической страницы
`-s строка`	Добавить указанную строку в конец каждого номера строки, чтобы отделить его от текста строки. По умолчанию используется один символ табуляции
`-v число`	Номер первой строки на каждой логической странице. По умолчанию имеет значение 1
`-w ширина`	Ширина поля номера строки. По умолчанию имеет значение 6

Следует отметить, что на практике нумеровать строки приходится довольно редко, но мы можем использовать `nl`, чтобы посмотреть, как объединить несколько инструментов для решения более сложных задач. Возьмем за основу наши наработки, созданные в предыдущей главе для получения отчета о дистрибутивах

Linux. Поскольку далее мы будем использовать программу `nl`, включим в текст разметку, отделяющую заголовок/тело/нижний колонтитул. Для этого откройте в текстовом редакторе сценарий для `sed` из предыдущей главы, добавьте в него строки с разметкой, как показано ниже, и сохраните сценарий в файле с именем *distros-nl.sed*:

```
# Сценарий для sed, создающий отчет о дистрибутивах Linux

1 i\
\\:\\:\\:\
\
Linux Distributions Report\
\
Name             Ver.    Released\
----             ----    --------\
\\:\\:
s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
$ a\
\\:\
\
End Of Report
```

Новый сценарий вставляет разметку логических страниц для `nl` и добавляет нижний колонтитул в конец отчета. Обратите внимание, что нам пришлось удвоить обратные слеши в разметке, потому что `sed` обычно интерпретирует их как экранирующие символы.

Теперь выведем улучшенный отчет, объединив `sort`, `sed` и `nl`:

```
[me@linuxbox ~]$ sort -k 1,1 -k 2n distros.txt | sed -f distros-nl.sed | nl


       Linux Distributions Report

        Name    Ver.    Released
        ----    ----    --------
     1  Fedora  5       2006-03-20
     2  Fedora  6       2006-10-24
     3  Fedora  7       2007-05-31
     4  Fedora  8       2007-11-08
     5  Fedora  9       2008-05-13
     6  Fedora  10      2008-11-25
     7  SUSE    10.1    2006-05-11
     8  SUSE    10.2    2006-12-07
     9  SUSE    10.3    2007-10-04
```

```
    10  SUSE    11.0    2008-06-19
    11  Ubuntu  6.06    2006-06-01
    12  Ubuntu  6.10    2006-10-26
    13  Ubuntu  7.04    2007-04-19
    14  Ubuntu  7.10    2007-10-18
    15  Ubuntu  8.04    2008-04-24
    16  Ubuntu  8.10    2008-10-30


      End Of Report
```

Наш отчет является результатом объединения в конвейер нескольких команд. Сначала мы отсортировали список по названиям дистрибутивов и номерам версий (поля 1 и 2), затем обработали результат программой `sed`, добавив заголовок отчета (включая разметку логических страниц для `nl`) и нижний колонтитул. В заключение мы обработали результат с помощью программы `nl`, которая по умолчанию нумерует только строки в потоке текста, принадлежащие разделу с телом логической страницы.

Попробуйте повторить команду и поэкспериментировать с разными параметрами команды `nl`. Интересный результат, например, можно получить с помощью

```
nl -n rz
```

и

```
nl -w 3 -s ' '
```

fold — перенос строк после указанной длины

Перенос строк заключается в разрыве текстовых строк по указанной ширине. Подобно другим рассматриваемым командам, `fold` может принимать одно или несколько имен файлов или данные со стандартного ввода. Если передать команде `fold` простой поток текста, можно увидеть, как она действует:

```
[me@linuxbox ~]$ echo "The quick brown fox jumped over the lazy dog." | fold -w 12
The quick br
own fox jump
ed over the
lazy dog.
```

Здесь мы видим, как действует программа `fold`. Текст, посланный командой `echo`, был разбит на сегменты указанной в параметре `-w` ширины. В этом примере мы ограничили ширину строк 12 символами. Если ширина не указана, по умолчанию она принимается равной 80 символам. Обратите внимание, что строки были разбиты без учета границ слов. Добавив параметр `-s`, можно заставить

`fold` разбивать строки по последнему доступному пробелу перед достижением указанной ширины:

```
[me@linuxbox ~]$ echo "The quick brown fox jumped over the lazy dog." |
                       fold -w 12 -s
The quick
brown fox
jumped over
the lazy
dog.
```

fmt — простое форматирование текста

Программа `fmt` также позволяет выполнить перенос строк плюс кое-что еще. Она принимает файлы или данные со стандартного ввода и формирует абзацы в потоке текста. По сути, она заполняет и объединяет строки, сохраняя пустые строки и отступы.

Для демонстрации нам понадобится некий текст. Возьмем фрагмент из Info-страницы для `fmt`:

```
   `fmt' читает текст из файла, заданного аргументами FILE (или со
стандартного ввода, если аргументы отсутствуют), и выводит результат
в стандартный вывод.

   По умолчанию пустые строки, пробелы между словами и отступы
сохраняются в выводе; последующие строки с разными отступами не
объединяются; символы табуляции на входе заменяются соответствующим
числом пробелов и выводятся в таком виде.

   `fmt' старается разбивать строки по концам предложений и стремится
не разрывать строки после первого слова или перед последним
словом в предложении. "Конец предложения" определяется либо по концу
абзаца, либо по слову, завершающемуся любым из символов `.?!', за
которым следуют два пробела или символ перевода строки, любые
скобки или кавычки при этом игнорируются. Подобно TEX, `fmt' читает
"абзацы" целиком, прежде чем выполнить перенос строк; программа
использует вариант алгоритма, предложенного Дональдом Э. Кнутом
(Donald E. Knuth) и Михаэлем Ф. Плассом (Michael F. Plass) в статье
"Breaking Paragraphs Into Lines", `Software--Practice & Experience'
11, 11 (November 1981), 1119-1184.
```

Скопируйте этот текст в текстовый редактор и сохраните в файле с именем *fmt-info.txt*. Теперь представьте, что нам нужно переформатировать этот текст, уместив его в колонку шириной 50 символов. Решить эту задачу можно с помощью команды `fmt` и ее параметра `-w`:

```
[me@linuxbox ~]$ fmt -w 50 fmt-info.txt | head
   `fmt' читает текст из файла, заданного
   аргументами FILE (или со
стандартного ввода, если аргументы отсутствуют),
и выводит результат в стандартный вывод.
   По умолчанию пустые строки, пробелы между
   словами и отступы
сохраняются в выводе; последующие строки с
разными отступами не объединяются; символы
```

Результат не особенно впечатляет. Может быть, стоит прочитать этот текст, так как он объясняет происходящее:

> По умолчанию пустые строки, пробелы между словами и отступы сохраняются в выводе; последующие строки с разными отступами не объединяются; символы табуляции на входе заменяются соответствующим числом пробелов и выводятся в таком виде.

Итак, `fmt` сохраняет отступ в первой строке. К счастью, `fmt` имеет параметр, исправляющий это:

```
[me@linuxbox ~]$ fmt -cw 50 fmt-info.txt
   `fmt' читает текст из файла, заданного
аргументами FILE (или со стандартного ввода,
если аргументы отсутствуют), и выводит результат
в стандартный вывод.

   По умолчанию пустые строки, пробелы между
словами и отступы сохраняются в выводе;
последующие строки с разными отступами не
объединяются; символы табуляции на входе
заменяются соответствующим числом пробелов и
выводятся в таком виде.

   `fmt' старается разбивать строки по концам
предложений и стремится не разрывать строки
после первого слова или перед последним
словом в предложении. "Конец предложения"
определяется либо по концу абзаца, либо по
слову, завершающемуся любым из символов `.?!',
за которым следует два пробела или символ
перевода строки, любые скобки или кавычки при
этом игнорируются. Подобно TeX, `fmt' читает
"абзацы" целиком, прежде чем выполнить перенос
строк; программа использует вариант алгоритма,
предложенного Дональдом Э. Кнутом (Donald
```

```
E. Knuth) и Михаэлем Ф. Плассом (Michael
F. Plass) в статье "Breaking Paragraphs Into
Lines", `Software--Practice & Experience' 11,
11 (November 1981), 1119-1184.
```

Намного лучше. Добавив параметр `-c`, мы получили желаемый результат.

Программа `fmt` содержит несколько интересных параметров, которые перечислены в табл. 21.3.

Таблица 21.3. Параметры fmt

Параметр	Значение
`-c`	Включить режим *обработки края* (crown margin). В этом режиме сохраняется отступ первых строк абзаца. Последующие строки выравниваются по второй строке
`-p строка`	Форматировать только строки, начинающиеся со строки, указанной в аргументе. После форматирования содержимое аргумента добавляется в начало каждой переформатированной строки. Этот параметр можно использовать для форматирования текста комментариев в исходном коде. Например, любой сценарий или конфигурационный файл, где комментарии начинаются с символа `#`, можно обработать командой `fmt` с параметром `-p '# '`, чтобы отформатировать только комментарии. Пример приводится ниже
`-s`	Режим простой разбивки. В этом режиме выполняется только разбивка строк по указанной ширине. Короткие строки не объединяются. Этот режим можно использовать для форматирования исходного кода, когда объединение коротких строк нежелательно
`-u`	Нормировать пробелы. Этот параметр применяется для форматирования в стандартном «машинописном стиле», то есть когда слова отделяются одним пробелом, а предложения — двумя. Этот режим удобно использовать для удаления *выравнивающих пробелов*, то есть пробелов, добавленных с целью выравнивания текста по обоим краям
`-w ширина`	Форматировать текст по указанной ширине. По умолчанию используется ширина 75 символов. Обратите внимание: в действительности `fmt` форматирует строки немного короче, чем указано в аргументе, чтобы обеспечить сбалансированность ширины текста

Особый интерес представляет параметр `-p`. С его помощью можно форматировать выбранные фрагменты файла, гарантировав, что все отформатированные строки будут начинаться с одной и той же последовательности символов. Многие языки программирования поддерживают комментарии, начинающиеся с символа решетки (`#`), и такие комментарии можно форматировать с помощью этого параметра.

Давайте создадим файл, имитирующий исходный текст программы с комментариями:

```
[me@linuxbox ~]$ cat > fmt-code.txt
# Этот файл содержит код с комментариями.

# Эта строка - комментарий.
# За ней следует другая строка с комментарием.
# И еще одна.

Это не комментарий, а строка с кодом.
Еще одна строка с кодом.
И еще.
```

Файл примера содержит комментарии, начинающиеся со строки `# ` (символ `#` и пробел), и строки «кода». Теперь воспользуемся командой `fmt`, чтобы отформатировать комментарии и при этом не затронуть код:

```
[me@linuxbox ~]$ fmt -w 50 -p '# ' fmt-code.txt
# Этот файл содержит код с комментариями.

# Эта строка - комментарий. За ней следует другая
# строка с комментарием. И еще одна.

Это не комментарий, а строка с кодом.
Еще одна строка с кодом.
И еще.
```

Обратите внимание, что смежные строки комментариев были объединены, а пустые строки и строки, не начинающиеся с указанного префикса, остались нетронутыми.

pr — форматирование текста для печати

Программа `pr` используется для разбивки текста на страницы. Перед печатью текст зачастую желательно разбить на страницы с несколькими пустыми строками в начале и в конце, чтобы получить пустые поля вверху и внизу каждой страницы. В дальнейшем эти поля можно использовать для вставки верхнего и нижнего колонтитулов.

Продемонстрируем работу `pr`, форматируя наш файл *distros.txt* в последовательность очень коротких страниц (ниже показаны только первые две страницы):

```
[me@linuxbox ~]$ pr -l 15 -w 65 distros.txt


2016-12-11 18:27              distros.txt              Page 1


SUSE            10.2    12/07/2006
Fedora          10      11/25/2008
SUSE            11.0    06/19/2008
Ubuntu          8.04    04/24/2008
Fedora          8       11/08/2007





2012-16-11 18:27              distros.txt              Page 2


SUSE            10.3    10/04/2007
Ubuntu          6.10    10/26/2006
Fedora          7       05/31/2007
Ubuntu          7.10    10/18/2007
Ubuntu          7.04    04/19/2007
```

В этом примере использовались параметры `-l` (длина (length) страницы) и `-w` (ширина (width) страницы), определяющие размеры «страницы» — 65 символов в ширину и 15 строк в длину. `pr` разбила содержимое файла *distros.txt* на отдельные страницы, добавив несколько пустых строк сверху и снизу, и создала заголовок по умолчанию, содержащий время последнего изменения файла, имя файла и номер страницы. Программа `pr` поддерживает множество параметров для управления форматированием страницы, но подробнее о них мы поговорим в главе 22.

printf — форматирование и вывод данных

В отличие от других команд в этой главе, команда `printf` не используется в конвейерной обработке (она не принимает данные со стандартного ввода) и редко применяется непосредственно в командной строке (чаще она используется в сценариях). Почему это так важно? Потому что она используется очень широко.

Команда `printf` (ее название происходит от *print formatted* (форматированный вывод)) первоначально была создана как функция для языка программирования C

и впоследствии была реализована во многих других языках, включая язык командной оболочки. Фактически в bash команда printf реализована как встроенная команда. Она имеет следующий синтаксис:

```
printf "формат" аргументы
```

Команда принимает строку с описанием формата, которая затем применяется к списку аргументов. Отформатированный результат передается в стандартный вывод. Ниже приводится простой пример:

```
[me@linuxbox ~]$ printf "I formatted the string: %s\n" foo
I formatted the string: foo
```

Строка формата может содержать литеральный текст (такой, как I formatted the string:), экранированные последовательности (такие, как \n, символ перевода строки) и последовательности, начинающиеся с символа %, которые называют *спецификаторами преобразований* (conversion specifications). В примере выше спецификатор преобразования %s используется для форматирования строки foo и включения ее в вывод команды. Еще один пример:

```
[me@linuxbox ~]$ printf "I formatted '%s' as a string.\n" foo
I formatted 'foo' as a string.
```

Как видите, в выводе команды спецификатор преобразования %s замещается строкой foo. Преобразование s используется для форматирования строковых данных. Существуют также другие спецификаторы для других видов данных. В табл. 21.4 перечислены наиболее часто используемые типы данных.

Таблица 21.4. Спецификаторы типов данных, наиболее часто используемых в команде printf

Спецификатор	Описание
d	Форматирует число как десятичное целое со знаком
f	Форматирует и выводит вещественное число
o	Форматирует целочисленное значение как восьмеричное число
s	Форматирует строку
x	Форматирует целочисленное значение как шестнадцатеричное число, с использованием букв *a*–*f* нижнего регистра, где это необходимо
X	То же, что и x, но использует буквы верхнего регистра
%	Выводит литеральный символ % (то есть сам спецификатор имеет вид: %%)

Продемонстрируем действие каждого спецификатора преобразования на примере строки `380`:

```
[me@linuxbox ~]$ printf "%d, %f, %o, %s, %x, %X\n" 380 380 380 380 380 380
380, 380.000000, 574, 380, 17c, 17C
```

Так как в строке формата указано шесть спецификаторов формата, нам потребовалось передать команде `printf` шесть аргументов. Шесть результатов показывают результат действия каждого спецификатора.

Для настройки вывода спецификатору формата можно передать несколько дополнительных компонентов. Полный синтаксис спецификатора преобразования выглядит так:

%[*флаги*][*ширина*][.*точность*]*спецификатор_преобразования*

Для правильной интерпретации дополнительные компоненты, если их несколько, должны передаваться в указанном порядке. Все компоненты описаны в табл. 21.5.

Таблица 21.5. Компоненты спецификаторов преобразований в команде printf

Спецификатор	Описание
`флаги`	Существует пять разных флагов: • `#` — использовать альтернативный формат вывода. Действует по-разному, в зависимости от типа данных. Преобразование `o` (восьмеричное число) добавляет в вывод префикс `0` (ноль). Преобразования `x` и `X` (шестнадцатеричное число) добавляют в вывод префикс `0x` или `0X` соответственно; • `0` **(ноль)** — добавляет нули в начало вывода. Это означает, что поле будет дополнено ведущими нулями, например: `000380`; • `-` **(дефис)** — выравнивание по левому краю. По умолчанию `printf` выполняет выравнивание по правому краю; • `' '` **(пробел)** — добавляет ведущий пробел перед положительными числами; • `+` **(знак «плюс»)** — выводит знак перед положительными числами. По умолчанию `printf` выводит знаки только перед отрицательными числами
`ширина`	Число, определяющее минимальную ширину поля вывода в символах
`.точность`	Определяет число знаков после десятичной запятой при выводе вещественных чисел. Для строковых значений точность определяет число выводимых символов

В табл. 21.6 перечислены некоторые примеры применения разных форматов.

Таблица 21.6. Примеры применения спецификаторов преобразований команды printf

Аргумент	Формат	Результат	Примечание
380	"%d"	380	Простое форматирование целых чисел
380	"%#x"	0x17c	Форматирование целочисленных значений в шестнадцатеричное представление с использованием альтернативного форматирования
380	"%05d"	00380	Форматирование целочисленных значений с ведущими нулями и минимальным размером поля, равным пяти символам
380	"%05.5f"	380.00000	Форматирование вещественных значений с ведущими нулями и 5 знаками после запятой. Поскольку указанная минимальная ширина поля (5) меньше фактической ширины отформатированного числа, ведущие нули не были добавлены
380	"%010.5f"	0380.00000	Ширина поля вывода увеличена до 10, вследствие чего появился ведущий нуль
380	"%+d"	+380	Флаг + требует выводить знак у положительных чисел
380	"%-d"	380	Флаг - обеспечивает форматирование с выравниванием по левому краю
abcdefghijk	"%5s"	abcdefghijk	Форматирование строки в поле с указанной минимальной шириной
abcdefghijk	"%.5s"	abcde	Применение компонента точности к строке привело к ее усечению

И еще раз: команда `printf` в основном используется в сценариях, где применяется для форматирования табличных данных, а не как самостоятельный инструмент командной строки. Тем не менее мы можем использовать ее для решения различных задач форматирования. Во-первых, попробуем вывести несколько полей, разделив их символами табуляции:

```
[me@linuxbox ~]$ printf "%s\t%s\t%s\n" str1 str2 str3
str1    str2    str3
```

Добавив `\t` (экранированную последовательность, соответствующую символу табуляции), мы достигли желаемого эффекта. Затем попробуем вывести несколько чисел в форматированном виде:

```
[me@linuxbox ~]$ printf "Line: %05d %15.3f Result: %+15d\n" 1071 3.14156295 32589
Line: 01071           3.142 Result:          +32589
```

Здесь демонстрируется действие компонента, определяющего минимальную ширину поля. А можно ли подобным образом отформатировать небольшую веб-страницу?

```
[me@linuxbox ~]$ printf "<html>\n\t<head>\n\t\t<title>%s</title>\n\t</head>
\n\t<body>\n\t\t<p>%s</p>\n\t</body>\n</html>\n" "Page Title" "Page Content"
<html>
        <head>
                <title>Page Title</title>
        </head>
        <body>
                <p>Page Content</p>
        </body>
</html>
```

Системы форматирования документов

До сих пор мы исследовали простые инструменты форматирования текста. Они хорошо подходят для решения небольших и простых задач, но как быть с более сложными заданиями? Одна из причин большой популярности операционной системы Unix среди технических специалистов и научных работников (кроме мощной поддержки многозадачности и многопользовательского окружения для выполнения любых работ, связанных с разработкой программного обеспечения) состоит в наличии инструментов, которые можно использовать для создания самых разных документов, таких как научные и академические публикации. Фактически, как описывается в документации GNU, средства подготовки документов положительно сказались на разработке Unix:

> Первая версия Unix была создана на машине PDP-7, простаивавшей в Bell Labs. В 1971-м разработчики захотели заполучить PDP-11 для дальнейшей работы над операционной системой. Чтобы оправдать затраты на эту систему, они внесли предложение о реализации системы форматирования документов для патентного бюро в AT&T. Эта первая программа форматирования являлась переделкой программы *roff* Макиллроя (McIllroy), которую написал Д. Ф. Оссанна (J. F. Ossanna).

В области систем форматирования документов доминируют два основных семейства программ: уходящие корнями в оригинальную программу `roff`, включая

`nroff` и `troff`, и основанные на системе верстки Дональда Кнута $\TeX$ (произносится как «тек»). И да, буква «Е» в середине имени действительно смещена вниз.

Имя *roff* произошло от словосочетания «run off» (напечатать), как во фразе: «Я напечатал копию для вас». Программа `nroff` используется для форматирования документов перед выводом на устройства, использующие моноширинные шрифты, такие как алфавитно-цифровые терминалы и принтеры, действующие подобно пишущим машинкам. На момент появления программы такие устройства составляли подавляющее большинство устройств вывода, подключаемых к компьютерам. Позднее появилась программа `troff`, форматирующая документы для вывода на наборные устройства, используемые для производства «готовых к тиражированию» макетов. Большинство современных принтеров способны имитировать вывод таких наборных устройств. Семейство `roff` также включает ряд других программ для подготовки фрагментов документов. К их числу относятся `eqn` (для форматирования математических формул) и `tbl` (для форматирования таблиц).

Система $\TeX$ (в версии, готовой к эксплуатации) впервые появилась в 1989 году и, до определенной степени, заменила `troff` как инструмент для получения документов типографского качества. Мы не будем рассматривать систему $\TeX$ здесь, во-первых, из-за ее сложности (существуют целые книги, посвященные ей) и, во-вторых, из-за того, что в большинстве современных систем Linux она не устанавливается по умолчанию.

СОВЕТ

Если у вас появится желание установить $\TeX$, обратите внимание на пакет `texlive`, присутствующий в репозиториях большинства дистрибутивов, а также на редактор графического содержимого LyX.

---

groff

`groff` — это пакет программ с GNU-реализацией `troff`. Он также включает сценарий, имитирующий работу `nroff`, и остальные программы семейства `roff`.

Семейство `roff` и его потомки использовались для создания форматированных документов способом, довольно чуждым современным пользователям. Большинство документов в наше время создается в текстовых процессорах, способных осуществлять составление и оформление документов в один шаг. До появления графических текстовых процессоров создание документов обычно происходило в два этапа. Сначала в текстовом редакторе выполнялось составление документа, а затем с помощью процессора, такого как `troff`, осуществлялось его форматирование.

Инструкции для программы форматирования встраивались в текст документа с применением языка разметки. Современным аналогом этого процесса может служить подготовка веб-страниц, которые записываются в текстовом редакторе и затем отображаются веб-браузером, интерпретирующим код HTML как инструкции языка разметки, описывающие окончательный вид страницы.

Мы не будем рассматривать семейство программ `groff` во всех подробностях, так как многие тонкости его языка разметки имеют прямое отношение к туманным для нас деталям типографского дела. Вместо этого мы сосредоточимся на одном из его *макропакетов*, широко использующемся до сих пор. Эти макропакеты упаковывают множество низкоуровневых команд в небольшое множество высокоуровневых команд, существенно упрощающих работу с `groff`.

Давайте ненадолго приостановимся и рассмотрим простую страницу справочного руководства (man). Она хранится в каталоге */usr/share/man* в виде текстового файла, сжатого с помощью `gzip`. Если заглянуть на распакованное содержимое, можно увидеть следующее (здесь показана страница справочного руководства из раздела 1 для команды `ls`):

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | head

.\" DO NOT MODIFY THIS FILE! It was generated by help2man 1.47.3.
.TH LS "1" "January 2018" "GNU coreutils 8.28" "User Commands"
.SH NAME
ls \- list directory contents
.SH SYNOPSIS
.B ls
[\fI\,OPTION\/\fR]... [\fI\,FILE\/\fR]...
.SH DESCRIPTION
.\" Add any additional description here
.PP
```

Если сравнить содержимое страницы с ее нормальным представлением на экране, можно заметить определенную корреляцию между языком разметки и результатом его интерпретации:

```
[me@linuxbox ~]$ man ls | head
LS(1)                     User Commands                     LS(1)



NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...
```

Этот пример интересен тем, что страницы справочного руководства отображаются программой `groff` с использованием макропакета `mandoc`. В действительности работу команды `man` можно сымитировать с помощью следующего конвейера.

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | groff -mandoc -T ascii | head
LS(1)                    User Commands                    LS(1)


NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...
```

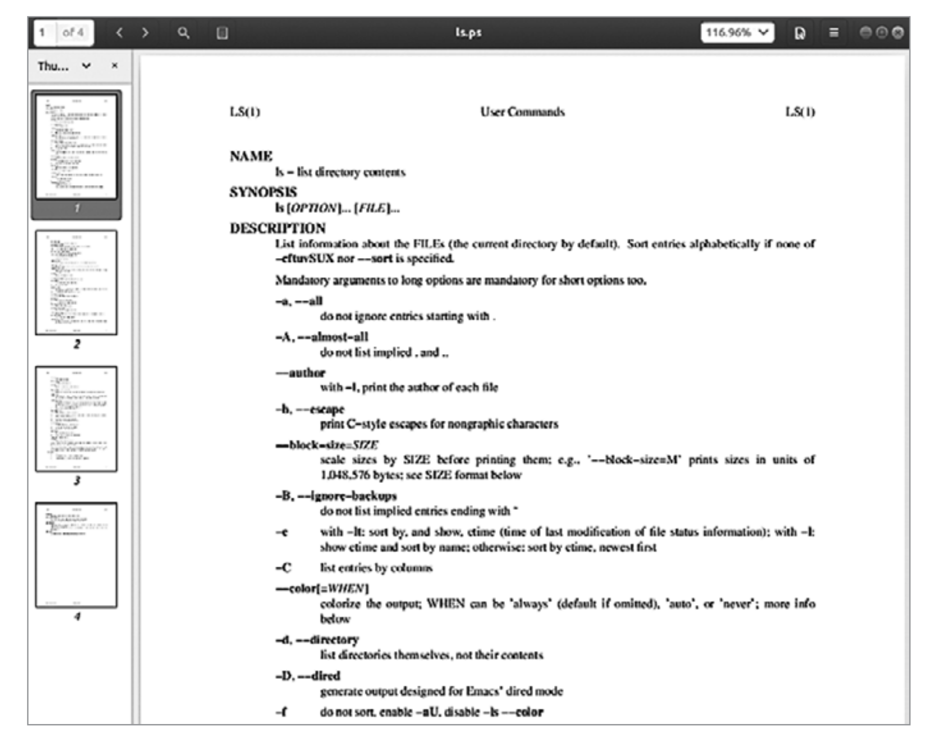
Здесь использована программа `groff` с множеством параметров, определяющих макропакет `mandoc` и драйвер вывода для ASCII. `groff` может выводить информацию в нескольких форматах. Если формат не задан, по умолчанию вывод производится в формате PostScript:

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | groff -mandoc | head
%!PS-Adobe-3.0
%%Creator: groff version 1.18.1
%%CreationDate: Thu Feb 2 13:44:37 2009
%%DocumentNeededResources: font Times-Roman
%%+ font Times-Bold
%%+ font Times-Italic
%%DocumentSuppliedResources: procset grops 1.18 1
%%Pages: 4
%%PageOrder: Ascend
%%Orientation: Portrait
```

Мы упоминали PostScript вскользь в предыдущей главе и снова вернемся к нему в следующей главе. PostScript — это *язык описания страниц*, используемый для вывода страниц на устройства печати с типографским качеством. Вывод команды можно сохранить в файл (здесь предполагается, что вы работаете в графическом окружении рабочего стола и в вашем домашнем каталоге имеется каталог *Desktop*):

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | groff -mandoc >
~/Desktop/foo.ps
```

После выполнения этой команды на рабочем столе появится значок файла. После двойного щелчка мышью на этом значке должен запуститься инструмент просмотра страниц и отобразить содержимое файла (рис. 21.1).


LS(1) User Commands LS(1)

NAME
ls – list directory contents

SYNOPSIS
ls [OPTION]... [FILE]...

DESCRIPTION
List information about the FILEs (the current directory by default). Sort entries alphabetically if none of –cftuvSUX nor ––sort is specified.

Mandatory arguments to long options are mandatory for short options too.

–a, ––all
do not ignore entries starting with .

–A, ––almost–all
do not list implied . and ..

––author
with –l, print the author of each file

–b, ––escape
print C–style escapes for nongraphic characters

––block–size=SIZE
scale sizes by SIZE before printing them; e.g., '––block–size=M' prints sizes in units of 1,048,576 bytes; see SIZE format below

–B, ––ignore–backups
do not list implied entries ending with ~

–c
with –lt: sort by, and show, ctime (time of last modification of file status information); with –l: show ctime and sort by name; otherwise: sort by ctime, newest first

–C
list entries by columns

––color[=WHEN]
colorize the output; WHEN can be 'always' (default if omitted), 'auto', or 'never'; more info below

–d, ––directory
list directories themselves, not their contents

–D, ––dired
generate output designed for Emacs' dired mode

–f
do not sort, enable –aU, disable –ls ––color

Рис. 21.1. Отображение содержимого PostScript-файла в окне обозревателя страниц в GNOME

Здесь мы видим прекрасно отформатированную страницу справочного руководства для команды `ls`! В действительности PostScript-файл можно преобразовать в формат *PDF* (*Portable Document Format* — формат переносимых документов), как показано ниже:

```
[me@linuxbox ~]$ ps2pdf ~/Desktop/foo.ps ~/Desktop/ls.pdf
```

Программа `ps2pdf` входит в состав пакета `ghostscript`, который устанавливается в большинстве систем Linux для поддержки печати.

СОВЕТ

Системы Linux включают множество программ командной строки для преобразования файлов из одного формата в другой. Они традиционно получают названия в соответствии с соглашением: *формат2формат*. Попробуйте выполнить команду `ls /usr/bin/*[[:alpha:]]2[[:alpha:]]*`, чтобы найти все такие программы. Также попробуйте поискать программы с именами *форматtoформат*.

В последнем упражнении с программой `groff` мы вновь вернемся к нашему старому доброму другу — файлу *distros.txt*. На этот раз мы воспользуемся программой `tbl`, которая применяется для форматирования таблиц, чтобы сформировать наш список дистрибутивов Linux. В этом упражнении мы задействуем сценарий для `sed`, созданный ранее, и с его помощью добавим разметку в поток текста, который затем передадим программе `groff`.

Сначала изменим сценарий для `sed`, добавив в него вызовы, необходимые программе `tbl`. Откройте сценарий *distros.sed* в текстовом редакторе и измените его, как показано ниже:

```
# Сценарий для sed, создающий отчет о дистрибутивах Linux

1 i\
.TS\
center box;\
cb s s\
cb cb cb\
l n c.\
Linux Distributions Report\
=\
Name    Version  Released\
_
s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/
$ a\
.TE
```

Обратите внимание, что для корректной работы сценария слова *Name Version Released* должны разделяться символами табуляции, а не пробелами. Сохраним получившийся файл с именем *distros-tbl.sed*. Программа `tbl` использует запросы `.TS` и `.TE` как метки начала и конца таблицы. Строки, следующие за запросом `.TS`, определяют глобальные свойства таблицы, в нашем случае выравнивание по центру страницы и окружение внешней рамкой. Остальные строки описывают форматирование строк таблицы. Если теперь снова запустить наш конвейер составления отчета с новым сценарием для `sed`, мы получим следующее:

```
[me@linuxbox ~]$ sort -k 1,1 -k 2n distros.txt | sed -f distros-tbl.sed |
            groff -t -T ascii
                 +------------------------------+
                 | Linux Distributions Report   |
                 +------------------------------+
                 | Name    Version    Released  |
                 +------------------------------+
                 |Fedora     5       2006-03-20 |
                 |Fedora     6       2006-10-24 |
```

```
|Fedora     7       2007-05-31 |
|Fedora     8       2007-11-08 |
|Fedora     9       2008-05-13 |
|Fedora    10       2008-11-25 |
|SUSE      10.1     2006-05-11 |
|SUSE      10.2     2006-12-07 |
|SUSE      10.3     2007-10-04 |
|SUSE      11.0     2008-06-19 |
|Ubuntu     6.06    2006-06-01 |
|Ubuntu     6.10    2006-10-26 |
|Ubuntu     7.04    2007-04-19 |
|Ubuntu     7.10    2007-10-18 |
|Ubuntu     8.04    2008-04-24 |
|Ubuntu     8.10    2008-10-30 |
+------------------------------+
```

Параметр `-t` требует от `groff` обработать поток текста с помощью `tbl`. Аналогично, параметр `-T` требует вывести данные в формате ASCII вместо используемого по умолчанию PostScript.

Это лучший формат вывода для тех, кто ограничен возможностями терминала или алфавитно-цифрового печатающего устройства. Если определить формат вывода PostScript и открыть результат в графическом обозревателе, мы получим намного более удовлетворительную картинку (рис. 21.2).

```
[me@linuxbox ~]$ sort -k 1,1 -k 2n distros.txt | sed -f distros-tbl.sed | groff -t
> ~/Desktop/foo.ps
```

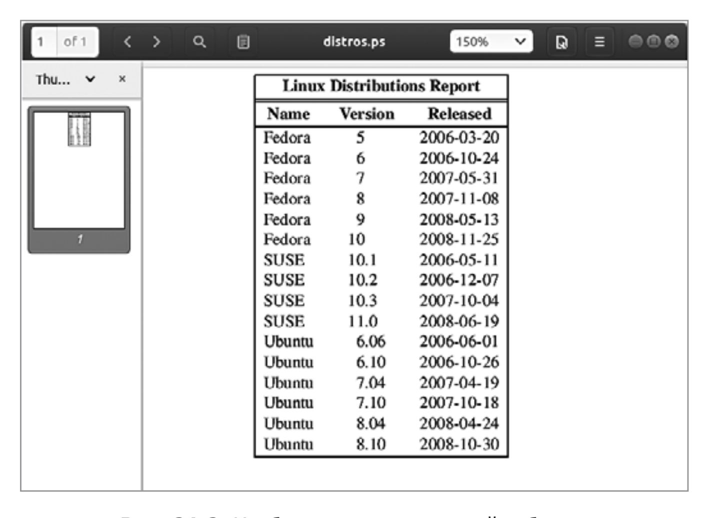


Рис. 21.2. Изображение законченной таблицы

Заключение

Учитывая, что текст занимает главенствующее положение в Unix-подобных операционных системах, было бы оправданно ожидать наличия большого числа инструментов для работы с текстом и его форматирования. И, как мы увидели, эти ожидания небеспочвенны! Простые инструменты форматирования, такие как `fmt` и `pr`, находят широкое применение в сценариях, производящих короткие документы, тогда как `groff` (с сопутствующими программами) можно использовать для создания книг. Возможно, вам никогда не придется использовать инструменты командной строки для написания технических статей (хотя многие делают это!), но знать о такой возможности вам не помешает.

22 Печать

После знакомства с приемами манипулирования текстом в двух предыдущих главах пришло время вывести его на бумагу. В этой главе мы рассмотрим инструменты командной строки, используемые для печати файлов и управления работой принтера. Мы не будем касаться вопросов настройки принтера, так как в разных дистрибутивах она осуществляется по-разному и обычно происходит автоматически в ходе установки. Для выполнения упражнений из этой главы вам понадобится настроенный и действующий принтер.

Далее мы обсудим следующие команды:

`pr` — преобразует текстовые файлы для печати;

`lpr` — печатает файлы;

`a2ps` — форматирует файлы для печати на принтере с поддержкой PostScript;

`lpstat` — выводит информацию о состоянии принтера;

`lpq` — выводит информацию о состоянии очереди печати;

`lprm` — отменяет задания печати.

Краткая история поддержки печати

Для полного понимания особенностей печати в Unix-подобных операционных системах сначала познакомимся с ее историей. Поддержка печати в Unix-подобных системах восходит к временам зарождения самой операционной системы. В те времена принтеры и способы их использования существенно отличались от современных.

Печать в ночное время

Подобно компьютерам, принтеры в эпоху, предшествующую появлению персональных компьютеров, были большими, дорогими и централизованными. Типичный пользователь 1980-х работал за терминалом на порядочном удалении от ЭВМ. Принтер же размещался рядом с ЭВМ и находился под неусыпным наблюдением операторов.

Когда принтеры были дорогими и централизованными, что было обычно в ранний период развития Unix, нормальной практикой считалось совместное использование принтера несколькими пользователями. Чтобы идентифицировать принадлежность задания печати определенному пользователю, в начале каждого задания печаталась *титульная страница* с именем пользователя. Персонал, обслуживающий ЭВМ, извлекал корзину с напечатанными за день документами и доставлял их конкретным пользователям.

Символьные принтеры

Технология печати в 80-х существенно отличалась от современной двумя аспектами. Во-первых, практически все принтеры того периода были *ударного действия*. В этих принтерах использовался механический узел, ударявший по красящей ленте, находящейся между узлом и бумагой и таким способом формировавший отпечатки символов на бумаге. В то время были наиболее распространены два вида печатающих механизмов: *лепестковый литероноситель* («ромашка») и *матричная печатающая головка*.

Во-вторых, что особенно важно, ранние принтеры использовали фиксированный набор символов, встроенный непосредственно в устройство. Например, принтер с «ромашкой» мог печатать только символы, присутствующие на лепестках «ромашки». В результате такие принтеры действовали как высокоскоростные печатающие машинки. Как большинство печатающих машинок, они печатали моноширинными шрифтами (шрифтами с фиксированной шириной символов). Это означает, что все символы имели одинаковую ширину. Печать символов выполнялась в фиксированных позициях на бумаге, и область печати имела фиксированное число знакомест. Большинство принтеров выводило 10 символов на дюйм (Characters Per Inch, CPI) по горизонтали и 6 строк на дюйм (Lines Per Inch, LPI) по вертикали. В соответствии с этой схемой на странице формата US-letter помещалось 85 символов в ширину и 66 строк в высоту. С учетом отступов слева и справа считалось, что максимальная ширина печати составляет 80 символов в строке. Это объясняет, почему дисплеи терминалов (и наших эмуляторов терминалов) обычно имеют ширину 80 символов. Таким способом обеспечивался режим «что вижу, то и получаю» (What You See Is What

You Get, WYSIWYG) отображения текста перед печатью с применением моноширинного шрифта.

Данные, посылаемые на принтер с ударным механизмом, — это простой поток байтов, соответствующих печатаемым символам. Например, чтобы напечатать *a*, посылался код символа 97 в наборе ASCII. Кроме того, в начале набора ASCII были выделены коды для управления кареткой принтера и бумагой, например, для возврата каретки в исходное положение, перевода строки, перевода формата (страницы) и др. С помощью управляющих кодов можно было имитировать некоторые эффекты, такие как жирная печать, когда принтеру посылался печатаемый символ, символ забоя (backspace) и снова тот же печатаемый символ, благодаря чему на бумаге получалось более темное изображение. В этом легко убедиться, если с помощью `nroff` отобразить страницу справочного руководства и исследовать вывод с помощью команды `cat -A`:

```
[me@linuxbox ~]$ zcat /usr/share/man/man1/ls.1.gz | nroff -man | cat -A | head
LS(1)                         User Commands                         LS(1)
$
$
$
N^HNA^HAM^HME^HE$
       ls - list directory contents$
$
S^HSY^HYN^HNO^HOP^HPS^HSI^HIS^HS$
       l^Hls^Hs [_^HO_^HP_^HT_^HI_^HO_^HN]... [_^HF_^HI_^HL_^HE]...$
```

Символы `^H` (CTRL-H) — это символы забоя (backspace), используемые для создания эффекта жирной печати. Аналогично для получения эффекта подчеркивания можно использовать последовательность из символов забоя/подчеркивания.

Графические принтеры

Создание графического интерфейса пользователя (GUI) привело к существенным изменениям в технологии печати. Так же как компьютеры все больше смещались в сторону использования графических дисплеев, печать все дальше уходила от символьных технологий вывода к графическим. Эта задача упростилась с появлением недорогих лазерных принтеров, которые вместо фиксированного набора символов могли осуществлять печать маленькими точками в любом месте области печати на странице. Это позволило использовать для печати пропорциональные шрифты (подобные тем, что применяются в книгопечатании) и даже печатать фотографии и высококачественные диаграммы.

Однако переход от символьной системы печати к графической вызвал появление огромных технических проблем. И вот почему. Число байтов, которое нужно

послать символьному принтеру для заполнения страницы, можно было подсчитать с помощью простой формулы (если предположить, что на странице умещается 60 строк, по 80 символов в каждой):

$60 \times 80 = 4800$ байт.

Для сравнения: лазерному принтеру с качеством печати 300 точек на дюйм (Dot Per Inch, DPI) для покрытия страницы размером 8 × 10 дюймов (203 × 254 мм) нужно послать

$(8 \times 300) \times (10 \times 300) \times 8 = 900\,000$ байт.

Многие медленные сети персональных компьютеров просто не могли достаточно быстро пропустить почти 1 мегабайт данных, чтобы напечатать на лазерном принтере полную страницу, поэтому требовалось какое-то новое решение.

Таким решением стало изобретение *языка описания страниц*. Язык описания страниц (Page Description Language, PDL) — это язык программирования, описывающий содержимое страницы. Программы на этом языке как бы говорили: «перейти в эту позицию, нарисовать символ *a* шрифтом Helvetica с кеглем 10 пунктов, перейти в эту позицию...», пока вся страница не была описана. Первым основным языком PDL стал *PostScript*, разработанный в Adobe Systems, он все еще широко используется в наше время. Язык PostScript — это полноценный язык программирования, ориентированный на книгопечатание и создание разного вида графических изображений. Он включает поддержку 35 стандартных высококачественных шрифтов плюс может принимать определения дополнительных шрифтов во время выполнения. На первом этапе поддержка PostScript встраивалась непосредственно в принтеры. Это решало проблему передачи данных. Даже притом, что типичная программа на PostScript по объему превышала простой поток байтов для символьных принтеров, ее размер был намного меньше числа байтов, необходимых для представления целой страницы.

Принтер с поддержкой PostScript принимал на входе программу на PostScript. Принтер имел собственный процессор и память (нередко принтеры имели большую вычислительную мощность, чем компьютеры, к которым они подключались) и выполнял специальную программу, называвшуюся *интерпретатором PostScript*, которая читала входящую программу на PostScript и *отображала* результат во внутреннюю память принтера, таким образом формируя шаблон из битов (точек) для вывода на бумагу. Такой процесс отображения чего-то в большой битовый шаблон (его называют *bitmap — растр*) в общем случае называют *процессором растровых изображений* (Raster Image Processor, RIP).

Спустя годы компьютеры и сети стали намного быстрее. Это позволило переместить RIP с принтера в компьютер, что, в свою очередь, позволило удешевить высококачественные принтеры.

Многие современные принтеры все еще способны принимать потоки символов, но большинство уже не поддерживают эту возможность. Они полагаются на RIP компьютера и ожидают получить поток битов для печати их в виде точек. Существуют также современные PostScript-принтеры.

Печать в Linux

Современные системы Linux используют два комплекта программного обеспечения для печати. Первый, CUPS (Common Unix Printing System — общая система печати для Unix), включает драйверы печати и средства управления заданиями; второй, Ghostscript, интерпретатор PostScript, действует как RIP.

Пакет CUPS осуществляет управление принтерами, создавая *очереди печати* и управляя ими. Как говорилось выше, в краткой исторической справке, поддержка печати в Unix первоначально была ориентирована на управление централизованным принтером, совместно используемым несколькими пользователями. Поскольку принтеры — довольно медленные устройства по своей природе в сравнении с компьютерами, которые поставляют им данные, системы печати должны обладать возможностью планирования множества заданий печати и их организации. Система CUPS также способна распознавать данные разных типов (в разумных пределах) и преобразовывать файлы в вид, пригодный для печати.

Подготовка файлов к печати

Так как мы — пользователи командной строки, наибольший интерес для нас представляет печать текста, хотя при этом сохраняется возможность печатать данные других форматов.

pr — преобразование текстовых файлов для печати

Мы уже касались программы `pr` в предыдущей главе. А теперь исследуем все богатство ее параметров, используемых при печати. В краткой исторической справке развития технологий печати рассказывалось, что символьные принтеры использовали мноноширинные шрифты, что обеспечивало фиксированное число символов в строке и строк на странице. Программа `pr` используется для выравнивания текста в соответствии с заданным размером страницы, с учетом дополнительных заголовков и полей на странице. Наиболее часто используемые параметры перечислены в табл. 22.1.

Таблица 22.1. Наиболее часто используемые параметры команды pr

Параметр	Описание
`+первая[:последняя]`	Вывести диапазон страниц, начиная со страницы с номером `первая` и заканчивая страницей с номером `последняя` (если указано)
`-колонок`	Вывести содержимое страницы в указанное число колонок
`-a`	По умолчанию, когда задан вывод в несколько колонок, колонки организованы по вертикали. Параметр `-a` (across — поперек) позволяет организовать колонки по горизонтали
`-d`	Вывести с двойным интервалом
`-D формат`	Формат вывода даты в заголовке страницы. Описание строки формата можно найти в странице справочного руководства (`man`) для команды `date`
`-f`	Использовать символ перевода формата вместо возврата каретки для отделения страниц друг от друга
`-h заголовок`	Текст для вывода в центре заголовка страницы вместо имени файла
`-l длина`	Длина страницы. По умолчанию длина устанавливается равной 66 строкам (соответствует формату US-letter с плотностью печати 6 строк на дюйм)
`-n`	Нумеровать строки
`-o отступ`	Создать левое поле, выполнив `отступ` указанного размера (в символах)
`-w ширина`	Ширина страницы в символах. По умолчанию `ширина` устанавливается равной 72 символам

Программа `pr` часто используется в конвейерах в роли фильтра. Следующий пример создает список содержимого каталога */usr/bin* и с помощью `pr` выводит его в три колонки с разбивкой на страницы:

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 -w 65 | head

2012-02-18 14:00                                       Page 1
[                     apturl                bsd-write
411toppm              ar                    bsh
a2p                   arecord               btcflash
```

```
a2ps                 arecordmidi          bug-buddy
a2ps-lpr-wrapper     ark                  buildhash
```

Отправка задания печати на принтер

Пакет программ печати CUPS поддерживает два метода печати, исторически используемых в Unix-подобных системах. Первый метод, с названием Berkeley, или LPD (используется в BSD-версиях Unix), основан на использовании программы `lpr`; второй метод, с названием SysV (используется в версиях Unix System V), основан на использовании программы `lp`. Обе программы работают примерно одинаково. Выбор той или иной зависит от личных предпочтений.

lpr — печать файлов (в стиле Berkeley)

Программа `lpr` применяется для отправки файлов на принтер. Она также может использоваться в конвейерах, так как способна принимать исходные данные со стандартного ввода. Например, напечатать предыдущий результат форматирования содержимого каталога в несколько колонок можно было бы так:

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 | lpr
```

В этом случае отчет будет выведен на принтер, используемый системой по умолчанию. Для вывода файла на другой принтер используйте параметр `-P`:

```
lpr -P имя_принтера
```

где аргумент `имя_принтера` — имя требуемого принтера. Получить список принтеров, известных системе, поможет команда

```
[me@linuxbox ~]$ lpstat -a
```

СОВЕТ

Многие дистрибутивы Linux позволяют определять фиктивные «принтеры», которые выводят результат в файл в формате PDF, а не на физический принтер. Это очень удобно для экспериментов с командами печати. Запустите свою программу настройки принтеров и посмотрите, поддерживает ли она такую возможность. Чтобы включить ее, в некоторых дистрибутивах может понадобиться установить дополнительные пакеты (такие, как `cups-pdf`).

В табл. 22.2 перечислены некоторые часто используемые параметры команды `lpr`.

Таблица 22.2. Наиболее часто используемые параметры команды lpr

Параметр	Описание
`-# число`	Число копий
`-p`	Вывести на каждой странице заголовок с датой, временем, именем задания и номером страницы. Этот так называемый параметр структурной печати («pretty print») можно использовать для печати текстовых файлов
`-P принтер`	Имя принтера для вывода. Если принтер не указан, используется системный принтер по умолчанию
`-r`	Удалить файлы после печати. Может пригодиться при использовании программ, создающих временные файлы для печати

lp — печать файлов (в стиле System V)

Подобно `lpr`, программа `lp` принимает файлы или данные со стандартного ввода. Отличается от `lpr` поддержкой иного (немного более сложного) набора параметров. Наиболее часто используемые параметры перечислены в табл. 22.3.

Таблица 22.3. Наиболее часто используемые параметры команды lp

Параметр	Описание
`-d принтер`	Имя принтера для вывода. Если принтер не указан, используется системный принтер по умолчанию
`-n число`	Число копий
`-o landscape`	Вывести в альбомной ориентации
`-o fitplot`	Масштабировать файл, чтобы уместить на странице. Этот параметр может пригодиться при печати изображений, например, файлов в формате JPEG
`-o scaling=число`	Масштабировать файл с указанным коэффициентом. Значение 100 соответствует полному заполнению страницы. Значения меньше 100 уменьшают размеры области печати, тогда как значения больше 100 — увеличивают, вследствие чего файл печатается на нескольких страницах
`-o cpi=число`	Установить плотность печати символов на дюйм (CPI) как указанное число символов на дюйм. По умолчанию это значение устанавливается равным 10
`-o lpi=число`	Установить плотность печати строк на дюйм (LPI) как указанное число строк на дюйм. По умолчанию это значение устанавливается равным 6

Параметр	Описание
`-o page-bottom=пунктов` `-o page-left=пунктов` `-o page-right=пунктов` `-o page-top=пунктов`	Размеры полей. Значения выражаются в *пунктах*, единице измерения, используемой в типографском деле. Один дюйм соответствует 72 пунктам
`-P страницы`	Список страниц для печати. Список может иметь вид перечисления номеров страниц через запятую и/или диапазонов, например: `1,3,5,7-10`

Вернемся к нашему списку содержимого каталога, но на этот раз выведем его с плотностью печати 12 CPI и 8 LPI и с левым полем размером полдюйма. Обратите внимание, что нам пришлось откорректировать параметры `pr`, чтобы учесть новые размеры страницы:

```
[me@linuxbox ~]$ ls /usr/bin | pr -4 -w 90 -l 88 | lp -o page-left=36 -o
cpi=12 -o lpi=8
```

Этот конвейер выводит список в четыре колонки с меньшим размером шрифта, чем принято по умолчанию. Увеличение плотности символов на дюйм позволило уместить больше колонок на странице.

Еще одна возможность: a2ps

Программа `a2ps` довольно интересна. Как можно догадаться по ее имени, это программа преобразования одного формата в другой, но не только. Первоначально ее имя означало ASCII to PostScript (из ASCII в PostScript) и она использовалась для подготовки текстовых файлов к печати на принтерах с поддержкой PostScript. С годами, однако, возможности программы росли, и теперь ее имя означает Anything to PostScript (все что угодно — в PostScript). Несмотря на то что имя программы говорит, что это — программа преобразования одного формата в другой, в действительности она является программой печати. Она выводит результаты своей работы в свой вывод по умолчанию — в системный принтер, а не в стандартный вывод. По умолчанию программа действует как программа «структурной печати», улучшая формат вывода. Мы можем с ее помощью создать PostScript-файл на своем рабочем столе:

```
[me@linuxbox ~]$ ls /usr/bin | pr -3 -t | a2ps -o ~/Desktop/ls.ps -L 66
[stdin (plain): 11 pages on 6 sheets]
[Total: 11 pages on 6 sheets] saved into the file `/home/me/Desktop/ls.ps'
```

Здесь мы обработали поток с помощью программы `pr`, передав ей параметр `-t` (чтобы опустить верхние и нижние колонтитулы), и передали результат программе

a2ps, указав ей файл для вывода (параметр -o) и плотность печати 66 строк на странице (параметр -L), чтобы разбить вывод pr на страницы. Если открыть получившийся файл с помощью соответствующего средства просмотра, можно увидеть, что он выглядит, как показано на рис. 22.1.

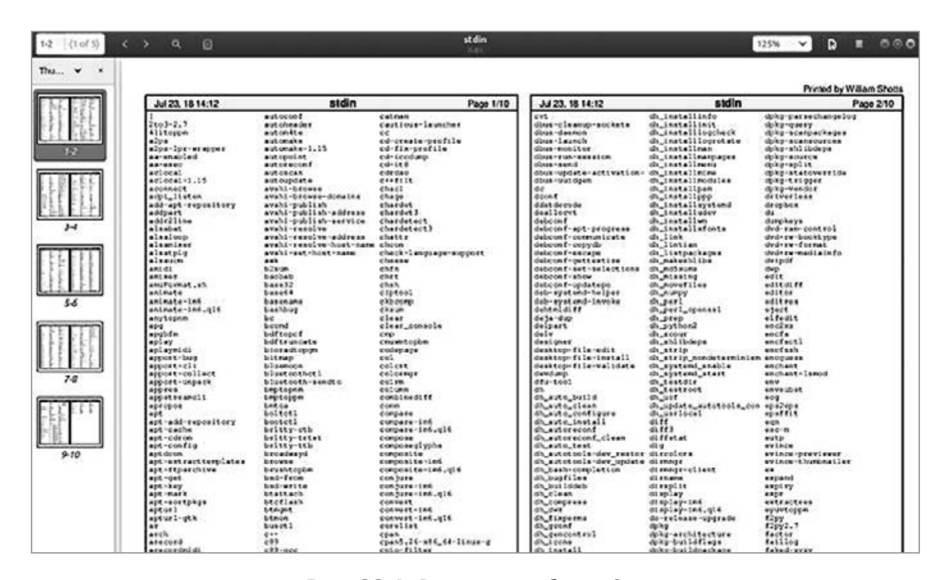


Рис. 22.1. Результат работы a2ps

Как видите, по умолчанию используется формат вывода «две страницы рядом». В этом формате содержимое двух страниц будет напечатано на одном листе бумаги. a2ps добавляет также свои хорошо отформатированные колонтитулы.

a2ps имеет множество параметров, они перечислены в табл. 22.4.

Таблица 22.4. Параметры команды a2ps

Параметр	Описание
--center-title текст	Текст для заголовка страницы в центре
--columns число	Число колонок для вывода страниц. По умолчанию 2
--footer текст	Текст для нижнего колонтитула
--guess	Вывести типы файлов, переданных программе в аргументах. Поскольку программа a2ps пытается преобразовывать и форматировать данные любых типов, этот параметр может пригодиться, чтобы понять, что a2ps будет делать с данным конкретным файлом

Параметр	Описание
--left-footer текст	Текст для нижнего колонтитула слева
--left-title текст	Текст для заголовка страницы слева
--line-numbers=интервал	Выводить номера строк через заданный интервал
--list=defaults	Вывести настройки по умолчанию
--pages диапазон	Печатать страницы из указанного диапазона
--right-footer текст	Текст для нижнего колонтитула справа
--right-title текст	Текст для заголовка страницы справа
--rows число	Разместить страницы в указанное число рядов. По умолчанию 1
-B	Не выводить заголовки страниц
-b текст	Текст заголовка страницы
-f размер	Использовать шрифт указанного размера
-l число	Число символов в строке. Этот параметр и параметр -L (ниже) можно использовать, чтобы правильно разместить на странице файлы, разбитые на страницы с помощью других программ, таких как pr
-L число	Число строк на странице
-M имя	Имя формата бумаги, например A4
-n число	Вывести указанное число копий каждой страницы
-o файл	Вывести результат в указанный файл. Если в качестве имени файла указан - (дефис), используется стандартный вывод
-P принтер	Имя принтера для вывода. Если принтер не указан, используется системный принтер по умолчанию
-R	Вывести в книжной ориентации
-r	Вывести в альбомной ориентации
-T число	Установить табулостопы через каждое указанное число символов
-u текст	Текст для нижнего слоя на странице («водяной знак»)

Это далеко не полный список. В действительности программа a2ps имеет намного больше параметров.

ПРИМЕЧАНИЕ

Кроме того, существует еще одна программа форматирования, которую можно использовать для преобразования текста в формат PostScript. Она называется `enscript` и способна выполнять почти те же виды форматирования и печати, что и `a2ps`, но, в отличие от последней, принимает только текстовые данные.

Наблюдение за заданиями печати и управление ими

Поскольку система печати в Unix изначально проектировалась для обработки заданий печати от нескольких пользователей, соответственно и система CUPS проектировалась исходя из той же предпосылки. Для каждого принтера создается своя *очередь печати*, в которой задания хранятся, пока не будут *переданы* принтеру. В составе CUPS имеется несколько программ командной строки для управления состоянием принтеров и очередей печати. Подобно программам `lpr` и `lp`, эти управляющие программы создавались после появления соответствующих программ из систем Berkeley и System V.

lpstat — вывод информации о состоянии принтера

Программу `lpstat` удобно использовать для определения имен и доступности принтеров в системе. Например, если к системе подключены два принтера — физический (с именем *printer*) и виртуальный, для вывода в файлы PDF (с именем *PDF*), — их состояние можно проверить так:

```
[me@linuxbox ~]$ lpstat -a
PDF accepting requests since Mon 05 Dec 2011 03:05:59 PM EST
printer accepting requests since Tue 21 Feb 2012 08:43:22 AM EST
```

Кроме того, с ее помощью можно получить более подробное описание конфигурации системы печати:

```
[me@linuxbox ~]$ lpstat -s
system default destination: printer
device for PDF: cups-pdf:/
device for printer: ipp://print-server:631/printers/printer
```

В этом примере видно, что имя *printer* соответствует системному принтеру по умолчанию и что это сетевой принтер, для взаимодействий с которым используется

протокол печати через Интернет (Internet Printing Protocol, *ipp://*), физически подключенный к системе с именем *print-server*.

В табл. 22.5 перечислены некоторые часто используемые параметры команды `lpstat`.

Таблица 22.5. Наиболее часто используемые параметры команды lpstat

Параметр	Описание
`-a [принтер...]`	Вывести состояние очереди печати для указанного принтера. Обратите внимание, что результат отражает состояние очереди печати для данного принтера и ее способность принимать задания, а не состояние физического принтера. Если принтер не указан, выводится информация обо всех очередях печати
`-d`	Вывести имя системного принтера по умолчанию
`-p [принтер...]`	Вывести состояние указанного принтера. Если принтер не указан, выводится информация обо всех принтерах
`-r`	Вывести состояние сервера печати
`-s`	Вывести сводную информацию о состоянии
`-t`	Вывести полный отчет о состоянии

lpq — вывод информации о состоянии очереди печати

Программа `lpq` используется для получения информации о состоянии очереди печати. С ее помощью можно увидеть состояние очереди и список заданий в ней. Ниже приводится пример вывода информации о состоянии пустой очереди для системного принтера по умолчанию с именем *printer*.

```
[me@linuxbox ~]$ lpq
printer is ready
no entries
```

Если принтер не указан (с помощью параметра `-P`), выводится информация об очереди для системного принтера по умолчанию. Если сформировать задание для печати и затем вывести информацию о состоянии очереди, это задание появится в списке:

```
[me@linuxbox ~]$ ls *.txt | pr -3 | lp
request id is printer-603 (1 file(s))
[me@linuxbox ~]$ lpq
```

```
printer is ready and printing
Rank    Owner   Job     File(s)         Total Size
active  me      603     (stdin)         1024 bytes
```

lprm и cancel — отмена заданий печати

В составе CUPS имеется две программы для завершения заданий печати и удаления их из очереди. Одна программа — в стиле Berkeley (`lprm`), а другая — в стиле System V (`cancel`). Они несколько отличаются поддерживаемыми параметрами, но, по сути, выполняют одну и ту же операцию. Если использовать пример с заданием печати, рассматриваемый выше, мы могли бы остановить выполнение задания и удалить его:

```
[me@linuxbox ~]$ cancel 603
[me@linuxbox ~]$ lpq
printer is ready
no entries
```

Обе команды имеют параметры, позволяющие удалить все задания, принадлежащие определенному пользователю задания, предназначенные для печати на определенном принтере, а также задания, содержащиеся в указанном списке номеров заданий. Все необходимые подробности вы найдете на страницах справочного руководства (man) для этих команд.

Заключение

В этой главе мы увидели, как в прошлом принтеры влияли на дизайн систем печати в Unix-подобных системах. Мы также выяснили, сколько имеется программ командной строки, помогающих не только планировать и выполнять задания печати, но также управлять различными параметрами вывода.

23 Компиляция программ

В этой главе мы посмотрим, как собирать программы, компилируя их исходный код. Доступность исходного кода — основное преимущество Linux, оно обеспечивает само существование этой системы. Вся экосистема разработки в Linux опирается на свободный обмен информацией между разработчиками. Для многих рядовых пользователей компиляция — утраченное искусство. Когда-то эта процедура была вполне обыденным делом, но в настоящее время создатели дистрибутивов поддерживают огромные репозитории с предварительно скомпилированными файлами, готовыми для загрузки и использования. На момент написания этих строк в репозитории дистрибутива Debian (одном из крупнейших) насчитывалось почти 68 000 пакетов.

Но зачем может понадобиться компилировать исходный код? Могу назвать две основные причины:

- **Доступность.** Несмотря на большое число предварительно скомпилированных пакетов в репозиториях дистрибутивов некоторые дистрибутивы могут включать не все необходимые приложения. В этом случае остается только один способ установить требуемую программу: скомпилировать ее из исходных кодов.
- **Своевременность.** Даже притом, что некоторые дистрибутивы специализируются на ультрасовременных версиях программ, многие все же немного отстают от прогресса. Это означает, что для получения самой последней версии программы придется ее скомпилировать.

Компиляция программ из исходных кодов может оказаться весьма специфическим и технически сложным делом, непосильным для некоторых пользователей. Однако многие программы компилируются относительно легко и просто, всего в несколько шагов. Все зависит от пакета. Далее мы рассмотрим очень простой случай, чтобы получить общее понимание процесса и начальные знания, отталкиваясь от которых желающие смогут продолжить исследования самостоятельно.

В этой главе будет представлена одна новая команда:

`make` — утилита сопровождения программ.

Что такое компиляция?

Выражаясь простым языком, компиляция — это процесс трансляции *исходного кода* (текста программы, описывающего ее действия и написанного программистом) на низкоуровневый язык, понятный процессору компьютера.

Процессор компьютера (Computer Processor Unit, CPU) работает на очень низком уровне, выполняя программы на языке, который называют *машинным*. Это числовой код, описывающий элементарные операции, такие как «сложить эти два байта», «сослаться на эту ячейку в памяти» или «скопировать этот байт». Каждая из этих инструкций выражается в двоичной форме (нулями и единицами). Самые первые программы писались на числовом коде, поэтому программисты, писавшие такой код, как поговаривают, много курили, пили кофе литрами и носили очки с толстенными линзами.

Эта проблема была решена с появлением *языка ассемблера*, который заменил числовые коды (слегка) более простыми символическими мнемониками, такими как CPY (для обозначения операции копирования) и MOV (для обозначения операции перемещения). Исходный код на языке ассемблера преобразовывался в машинный код программой, называющейся *ассемблером*. Язык ассемблера используется и в наши дни для решения специальных задач программирования, таких как разработка *драйверов устройств* или *встраиваемых систем*.

Затем появились *высокоуровневые языки программирования*. Они называются так потому, что позволяют программисту меньше думать об особенностях работы процессора и больше — о решении задачи, стоящей перед ним. К числу этих первых языков (разработанных в течение 1950-х) относятся: *FORTRAN* (создавался для решения научных и технических задач) и *COBOL* (для решения экономических задач). Оба продолжают ограниченно использоваться и по сию пору.

Несмотря на большое число популярных языков программирования, господствующие позиции занимают только два из них. Многие современные системы и программы написаны на C или на C++. В примерах ниже мы будем компилировать программы на языке C.

Программы на языках высокого уровня преобразуются в инструкции на машинном языке с помощью другой программы — *компилятора*. Некоторые компиляторы транслируют высокоуровневые инструкции на язык ассемблера, а затем

с помощью ассемблера производят окончательную трансляцию на машинный язык.

Компиляции, как правило, сопутствует процесс *компоновки*. Программы часто выполняют множество типовых операций. Возьмем для примера операцию открытия файла. Многие программы выполняют ее, но было бы слишком расточительно в каждой программе реализовывать свою процедуру открытия файлов. Предпочтительнее иметь единый программный код, знающий, как открывать файлы, и дать всем программам возможность использовать его. Поддержка решения типовых задач осуществляется с помощью *библиотек*. Они содержат множество *подпрограмм*, которые решают типовые задачи и могут использоваться множеством программ. Если заглянуть в каталоги */lib* и */usr/lib*, мы обнаружим подобные библиотеки. Для формирования связей между результатом работы компилятора и библиотеками, необходимыми компилируемой программе, используется *программа-компоновщик* (linker, ее также называют *редактором связей*). Окончательным результатом этого процесса является *выполняемый файл программы*, готовый к использованию.

Все ли программы компилируются?

Нет. Как мы уже видели, некоторые программы, такие как сценарии на языке командной оболочки, не требуют компиляции и выполняются непосредственно. Они написаны на языках, которые называют *языками сценариев*, или *интерпретируемыми языками*. К числу этих языков, популярность которых только растет в последние годы, относятся Perl, Python, PHP, Ruby и многие другие.

Программы на языках сценариев выполняются специальной программой, *интерпретатором*. Интерпретатор получает файл программы, читает его и выполняет каждую инструкцию, содержащуюся в нем. Вообще, интерпретируемые программы выполняются намного медленнее, чем компилируемые. Это объясняется необходимостью транслировать исходный код каждой инструкции в интерпретируемой программе всякий раз, когда она встречается, тогда как в скомпилированной программе исходный код инструкций был уже однажды преобразован в машинный код и сохранен в окончательном выполняемом файле.

Но почему тогда интерпретирующие языки так популярны? Для многих рутинных задач они оказываются «достаточно быстрыми», но самое большое их достоинство в простоте и скорости разработки интерпретируемых программ в сравнении с компилируемыми. Разработка программ — это обычно циклический процесс, включающий три этапа: создание исходного кода, компиляцию и тестирование. С увеличением программы в размерах этап компиляции в упомянутом цикле

может оказаться весьма продолжительным. Интерпретирующие языки избавляют от необходимости компиляции и тем самым ускоряют их разработку.

Компиляция программ на C

Давайте что-нибудь скомпилируем. Для этого нам понадобятся некоторые инструменты, такие как компилятор, компоновщик и утилита `make`. Практически во всех системах Linux используется один и тот же компилятор языка C с именем `gcc` (GNU C Compiler), первоначально написанный Ричардом Столлманом. Многие дистрибутивы не устанавливают `gcc` по умолчанию. Проверить его присутствие в системе можно так:

```
[me@linuxbox ~]$ which gcc
/usr/bin/gcc
```

Результат свидетельствует о присутствии компилятора.

СОВЕТ

Ваш дистрибутив может иметь метапакет (коллекцию пакетов) для разработки программ. В этом случае рекомендуется установить его, если вы собираетесь компилировать программы в своей системе. Если такой метапакет отсутствует, попробуйте установить пакеты `gcc` и `make`. Во многих дистрибутивах их вполне достаточно для выполнения упражнений, предлагаемых далее.

Получение исходного кода

В нашем упражнении мы скомпилируем программу с названием `diction` из проекта GNU. Эта маленькая удобная программка проверяет качество и стиль содержимого текстовых файлов. А поскольку она невелика, она легко компилируется.

Следуя соглашениям, мы сначала создадим каталог `src` для исходного кода и затем загрузим в него исходный код с помощью команды `ftp`:

```
[me@linuxbox ~]$ mkdir src
[me@linuxbox ~]$ cd src
[me@linuxbox src]$ ftp ftp.gnu.org
Connected to ftp.gnu.org.
220 GNU FTP server ready.
Name (ftp.gnu.org:me): anonymous
230 Login successful.
```

```
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd gnu/diction
250 Directory successfully changed.
ftp> ls
200 PORT command successful. Consider using PASV.
150 Here comes the directory listing.
-rw-r--r--    1 1003  65534   68940 Aug 28  1998 diction-0.7.tar.gz
-rw-r--r--    1 1003  65534   90957 Mar 04  2002 diction-1.02.tar.gz
-rw-r--r--    1 1003  65534  141062 Sep 17  2007 diction-1.11.tar.gz
226 Directory send OK.
ftp> get diction-1.11.tar.gz
local: diction-1.11.tar.gz remote: diction-1.11.tar.gz
200 PORT command successful. Consider using PASV.
150 Opening BINARY mode data connection for diction-1.11.tar.gz (141062
bytes).
226 File send OK.
141062 bytes received in 0.16 secs (847.4 kB/s)
ftp> bye
221 Goodbye.
[me@linuxbox src]$ ls
diction-1.11.tar.gz
```

В предыдущем примере мы использовали традиционный метод с применением `ftp`, но есть и другие способы загрузки исходного кода. Например, проект GNU поддерживает также загрузку по протоколу HTTPS. То есть мы могли бы загрузить исходный код `diction` с помощью программы `wget`:

```
[me@linuxbox src]$ wget https://ftp.gnu.org/gnu/diction/diction-1.11.tar.gz
--2018-07-25 09:42:20-- https://ftp.gnu.org/gnu/diction/diction-1.11.tar.gz
Resolving ftp.gnu.org (ftp.gnu.org)... 208.118.235.20, 2001:4830:134:3::b
Connecting to ftp.gnu.org (ftp.gnu.org)|208.118.235.20|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 141062 (138K) [application/x-gzip]
Saving to: 'diction-1.11.tar.gz'
diction-1.11.tar.gz 100%[===================>] 137.76K
--.-KB/s in 0.09s

2018-07-25 09:42:20 (1.43 MB/s) - 'diction-1.11.tar.gz.1' saved [141062/141062]
```

ПРИМЕЧАНИЕ

Поскольку мы сами управляем процессом компиляции исходного кода, поместим его в каталог *~/src*. Исходный код, устанавливаемый дистрибутивом, помещается в каталог */usr/src*, а исходный код, предназначенный для использования множеством пользователей, обычно устанавливается в */usr/local/src*.

Исходный код обычно распространяется в виде сжатого tar-файла. Иногда называемые *тарболлами* (tarball), эти файлы содержат дерево исходных текстов, или иерархию каталогов и файлов, составляющих исходный код. Подключившись к FTP-сайту, мы получили список доступных tar-файлов и выбрали для загрузки самую свежую версию. При помощи команды `get` программы `ftp` скопировали файл с сервера FTP на локальную машину.

После загрузки tar-файла его нужно распаковать. Делается это с помощью программы `tar`:

```
[me@linuxbox src]$ tar xzf diction-1.11.tar.gz
[me@linuxbox src]$ ls
diction-1.11        diction-1.11.tar.gz
```

СОВЕТ

Программа `diction`, подобно всем программам из проекта GNU, следует определенным стандартам упаковки исходного кода. Большая часть других исходных кодов, доступных в экосистеме Linux, также следует этому стандарту. Одним из элементов стандарта является создание каталога с деревом исходных текстов и именем `project-x.xx` после распаковывания tar-файла, то есть с именем, содержащим имя проекта и номер версии. Такая схема упрощает установку нескольких версий одной и той же программы. Однако перед распаковыванием хорошо бы исследовать организацию дерева. При распаковывании некоторых проектов каталог не создается, а файлы помещаются непосредственно в текущий каталог, что может вызвать неразбериху и путаницу в хорошо организованном каталоге *src*. Чтобы избежать этого, пользуйтесь следующей командой для исследования содержимого tar-файла:

```
tar tzvf tarfile | head
```

Исследование дерева исходных текстов

В результате распаковывания tar-файла был создан новый каталог *diction-1.11*. Этот каталог содержит дерево исходных текстов. Давайте заглянем внутрь:

```
[me@linuxbox src]$ cd diction-1.11
[me@linuxbox diction-1.11]$ ls
config.guess  diction.c       getopt.c      nl
config.h.in   diction.pot     getopt.h      nl.po
config.sub    diction.spec    getopt_int.h  README
configure     diction.spec.in INSTALL       sentence.c
configure.in  diction.texi.in install-sh    sentence.h
```

```
COPYING        en             Makefile.in   style.1.in
de             en_GB          misc.c        style.c
de.po          en_GB.po       misc.h        test
diction.1.in   getopt1.c      NEWS
```

Здесь мы видим множество файлов. Программы, принадлежащие проекту GNU, а также многие другие поставляются вместе с файлами документации *README*, *INSTALL*, *NEWS* и *COPYING*. В них содержится описание программы, информация о порядке сборки и установки и условия лицензионного соглашения. Я рекомендую всегда внимательно прочитывать файлы *README* и *INSTALL* перед сборкой программы.

Другими интересными файлами в этом каталоге являются файлы с расширениями *.c* и *.h*:

```
[me@linuxbox diction-1.11]$ ls *.c
diction.c  getopt1.c  getopt.c  misc.c  sentence.c  style.c
[me@linuxbox diction-1.11]$ ls *.h
getopt.h  getopt_int.h  misc.h  sentence.h
```

Файлы с расширением *.c* содержат две программы на C, входящие в состав пакета (`style` и `diction`), разбитые на несколько модулей. Большие программы часто разбивают на более мелкие, более простые в сопровождении фрагменты. Файлы с исходным кодом содержат простой текст, и их можно исследовать с помощью `less`:

```
[me@linuxbox diction-1.11]$ less diction.c
```

Файлы с расширением *.h* известны как *заголовочные файлы*. Они тоже содержат простой текст — описание подпрограмм, подключаемых из файла с исходным кодом или библиотеки. Чтобы компилятор смог связать модули, он должен иметь описание всех модулей, составляющих единую программу. Ближе к началу в файле *diction.c* имеется следующая строка:

```
#include "getopt.h"
```

Она требует от компилятора прочитать фал *getopt.h*, прежде чем продолжать чтение исходного кода в *diction.c*, чтобы «узнать», что имеется в файле *getopt.c*. Файл *getopt.c* содержит код подпрограмм, используемых обеими программами, `style` и `diction`.

Инструкции подключения файла *getopt.h* предшествует еще несколько инструкций `include`:

```
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

Они также ссылаются на заголовочные файлы, но эти файлы хранятся за пределами дерева исходных текстов. Они должны поставляться системой для поддержки компиляции программ. Эти файлы можно найти в каталоге */usr/include*:

```
[me@linuxbox diction-1.11]$ ls /usr/include
```

Заголовочные файлы помещаются в этот каталог в процессе установки компилятора.

Сборка программ

В большинстве случаев сборка программы заключается в выполнении последовательности из двух команд:

```
./configure
make
```

Программа `configure` — это сценарий командной оболочки, поставляемый вместе с деревом исходных текстов. Его задача — проанализировать окружение сборки. Большинство исходного кода поддерживает *переносимость*. То есть такой исходный код спроектирован так, что допускает сборку в разных Unix-подобных системах. Но для этого во время сборки в исходный код может потребоваться внести небольшие изменения, учитывающие различия между системами. Программа `configure` также проверяет наличие необходимых внешних инструментов и компонентов.

Давайте запустим `configure`. Так как эта программа находится не там, где командная оболочка обычно ищет выполняемые файлы, нужно явно сообщить ей местоположение программы, добавив в команду префикс `./`. Он указывает, что программа находится в текущем рабочем каталоге:

```
[me@linuxbox diction-1.11]$ ./configure
```

В процессе проверки и настройки сборки `configure` выведет множество сообщений. Последние строки ее вывода должны выглядеть примерно так:

```
checking libintl.h presence... yes
checking for libintl.h... yes
checking for library containing gettext... none required
configure: creating ./config.status
config.status: creating Makefile
config.status: creating diction.1
config.status: creating diction.texi
config.status: creating diction.spec
config.status: creating style.1
```

```
config.status: creating test/rundiction
config.status: creating config.h
[me@linuxbox diction-1.11]$
```

Самое важное здесь — отсутствие сообщений об ошибках. Их появление означало бы неудачу настройки и невозможность сборки программы до их устранения.

Как мы видим, configure создала в каталоге с исходным кодом несколько файлов. Самым важным является файл сборки *Makefile*. Файл сборки — это конфигурационный файл с инструкциями для программы make, описывающими, как собрать программу. Без такого файла утилита make работать не будет. Это обычный текстовый файл, то есть мы можем заглянуть в него:

```
[me@linuxbox diction-1.11]$ less Makefile
```

Программа make принимает файл сборки (обычно с именем *Makefile*), в котором описываются отношения и зависимости между компонентами, составляющими окончательную программу.

Первый раздел в файле сборки определяет переменные для подстановки в последующих его разделах. Например, здесь можно увидеть строку

```
CC=             gcc
```

определяющую, что роль компилятора С будет играть gcc. Далее в файле можно посмотреть, как используется это определение:

```
diction:        diction.o sentence.o misc.o getopt.o getopt1.o
                $(CC) -o $@ $(LDFLAGS) diction.o sentence.o misc.o \
                getopt.o getopt1.o $(LIBS)
```

Здесь выполняется подстановка: во время выполнения конструкция $(CC) замещается командой gcc.

Большую часть файла сборки занимают строки, определяющие *целевой файл* (target) — в данном случае выполняемый файл diction — и файлы, от которых он зависит. Остальные строки описывают команды, которые необходимо выполнить для создания целевого файла из его компонентов. Мы видим, что выполняемый файл diction (одна из конечных программ) зависит от присутствия файлов *diction.o*, *sentence.o*, *misc.o*, *getopt.o* и *getopt1.o*. Далее в файле сборки присутствуют определения, в которых каждый из этих файлов играет роль целевого.

```
diction.o:      diction.c config.h getopt.h misc.h sentence.h
getopt.o:       getopt.c getopt.h getopt_int.h
getopt1.o:      getopt1.c getopt.h getopt_int.h
```

```
misc.o:         misc.c config.h misc.h
sentence.o:     sentence.c config.h misc.h sentence.h
style.o:        style.c config.h getopt.h misc.h sentence.h
```

Однако в этих определениях не видно ни одной команды. Обработка этих строк осуществляется определением общей цели, что находится выше в файле, где описывается команда компиляции всех файлов с расширением *.c* в файлы с расширением *.o*:

```
.c.o:
                $(CC) -c $(CPPFLAGS) $(CFLAGS) $<
```

На первый взгляд все это кажется очень сложным. Почему бы просто не перечислить все этапы компиляции? Ответ на этот вопрос станет очевиден чуть позже. А пока давайте запустим `make` и соберем наши программы:

```
[me@linuxbox diction-1.11]$ make
```

Программа `make` запустится и выполнит все инструкции в файле *Makefile*. В процессе работы она выведет множество сообщений. А по завершении мы увидим, что в каталоге появились все целевые файлы:

```
[me@linuxbox diction-1.11]$ ls
config.guess   de.po            en            install-sh   sentence.c
config.h       diction          en_GB         Makefile     sentence.h
config.h.in    diction.1        en_GB.mo      Makefile.in  sentence.o
config.log     diction.1.in     en_GB.po      misc.c       style
config.status  diction.c        getopt1.c     misc.h       style.1
config.sub     diction.o        getopt1.o     misc.o       style.1.in
configure      diction.pot      getopt.c      NEWS         style.c
configure.in   diction.spec     getopt.h      nl           style.o
COPYING        diction.spec.in  getopt_int.h  nl.mo        test
de             diction.texi     getopt.o      nl.po
de.mo          diction.texi.in  INSTALL       README
```

Среди них `diction` и `style`, программы, которые мы намеревались собрать. Примите заслуженные поздравления! Мы только что скомпилировали первые программы из исходного кода!

Но, исключительно ради любопытства, запустим `make` еще раз:

```
[me@linuxbox diction-1.11]$ make
make: Nothing to be done for `all'[1].
```

[1] Для `` `all' `` ничего не нужно делать. — *Примеч. пер.*

Она вывела довольно странное сообщение. Но почему? Почему она не выполнила сборку программы повторно? Во всем виновата make. Вместо того чтобы просто собрать все заново, make собирает только то, что нужно собрать. Так как все целевые файлы уже присутствуют в каталоге, make решила, что ничего больше делать не требуется. Продемонстрировать это можно, удалив одну из собранных целей и запустив make снова:

```
[me@linuxbox diction-1.11]$ rm getopt.o
[me@linuxbox diction-1.11]$ make
```

Вы увидите, что make повторно собирает getopt.o и заново компонует программы diction и style, потому что они зависят от отсутствующего модуля. Такое поведение указывает на еще одну важную особенность make: она старается обеспечить актуальность целевых файлов. make гарантирует, что целевые файлы будут более новыми, чем их зависимости. В этом есть определенный смысл, потому что программист часто сначала изменяет исходный код, а затем запускает make, чтобы собрать новую версию программы. make гарантирует сборку всех целевых файлов, опирающихся на изменившийся код. Воспользуемся программой touch, чтобы «обновить» один из файлов с исходным кодом, и посмотрим, к чему это приведет:

```
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me      me      37164 2009-03-05 06:14 diction
-rw-r--r-- 1 me      me      33125 2007-03-30 17:45 getopt.c
[me@linuxbox diction-1.11]$ touch getopt.c
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me      me      37164 2009-03-05 06:14 diction
-rw-r--r-- 1 me      me      33125 2009-03-05 06:23 getopt.c
[me@linuxbox diction-1.11]$ make
```

Когда make завершится, мы увидим, что целевой файл стал «свежее» зависимости:

```
[me@linuxbox diction-1.11]$ ls -l diction getopt.c
-rwxr-xr-x 1 me      me      37164 2009-03-05 06:24 diction
-rw-r--r-- 1 me      me      33125 2009-03-05 06:23 getopt.c
```

Способность программы make выполнять сборку только целей, которые действительно этого требуют, дает программистам немалые выгоды. Экономия времени, возможно, не очевидна для нашего маленького проекта, но она намного заметнее в больших проектах. Вспомните, например, что ядро Linux (программа, которая постоянно изменяется и совершенствуется) содержит несколько *миллионов* строк кода.

Установка программ

Старательно упакованный исходный код часто включает специальную цель для `make`, которая называется `install` (установить). Эта цель выполняет установку готового программного продукта в системный каталог. Обычно это каталог */usr/local/bin*, традиционное место для установки программного обеспечения, собранного в локальной системе. Однако этот каталог, как правило, недоступен рядовым пользователям для записи, поэтому, чтобы выполнить установку, вам потребуются привилегии суперпользователя:

```
[me@linuxbox diction-1.11]$ sudo make install
```

После установки проверим готовность программы к использованию:

```
[me@linuxbox diction-1.11]$ which diction
/usr/local/bin/diction
[me@linuxbox diction-1.11]$ man diction
```

Все на месте!

Заключение

В этой главе мы узнали, как с помощью трех простых команд — `./configure`, `make`, `make install` — собирать пакеты из исходных кодов. Кроме того, мы увидели, насколько важную роль играет `make` в сопровождении программ. Программу `make` можно использовать для решения многих задач, где требуется поддерживать отношения цель/зависимость, а не только для компиляции исходного кода.

Часть IV

СЦЕНАРИИ КОМАНДНОЙ ОБОЛОЧКИ

24 Создание первого сценария командной оболочки

В предыдущих главах мы собрали арсенал инструментов командной строки. Несмотря на то что эти инструменты можно использовать для решения самых разных вычислительных задач, мы все еще вынуждены использовать их по старинке, по одному. Было бы замечательно переложить еще больше ручной работы на командную оболочку. Оказывается, это возможно. Объединяя инструменты в программы собственной конструкции, мы можем заставить командную оболочку самостоятельно выполнять сложные последовательности операций. Такие программы называются *сценариями командной оболочки*.

Что такое сценарии командной оболочки

Выражаясь простым языком, сценарий командной оболочки — это файл, содержащий последовательность команд. Командная оболочка читает этот файл и выполняет команды, как если бы они вводились вручную в командной строке.

Командная оболочка — это одновременно и мощный интерфейс командной строки к системе, и интерпретатор языка сценариев. Как вы увидите далее, многое из того, что можно сделать в командной строке, также можно сделать в сценариях, а многое из того, что можно сделать в сценариях, можно сделать в командной строке.

Мы уже познакомились с множеством особенностей командной оболочки, но все внимание уделялось только особенностям, связанным с непосредственным использованием командной строки. Но командная оболочка обладает также множеством особенностей, обычно (но не всегда) используемых при создании программ.

Как написать сценарий командной оболочки

Чтобы успешно создать и запустить сценарий командной оболочки, нам нужно:

1. **Написать сценарий.** Сценарии командной оболочки — это обычные текстовые файлы. Поэтому для их создания нам понадобится текстовый редактор. Лучше использовать текстовый редактор, обладающий функцией подсветки синтаксиса, позволяющей видеть элементы сценариев с цветной маркировкой. Подсветка синтаксиса помогает замечать некоторые типичные ошибки. Для создания сценариев хорошо подходят `vim`, `gedit`, `kate` и многие другие редакторы.
2. **Сделать сценарий выполняемым.** Система не позволяет интерпретировать любой старый текстовый файл как программу, и небезосновательно! Поэтому, чтобы выполнить сценарий, файлу сценария нужно дать разрешения на выполнение.
3. **Поместить сценарий в каталог, где командная оболочка сможет найти его.** Командная оболочка автоматически ищет выполняемые файлы в нескольких каталогах, если путь к файлу не указан явно. Для максимального удобства мы будем помещать наши сценарии в такие каталоги.

Формат файла сценария

Следуя традициям программирования, напишем программу «hello world», чтобы продемонстрировать чрезвычайно простой сценарий. Итак, запустите текстовый редактор и введите следующий сценарий:

```
#!/bin/bash
# Это наш первый сценарий.
echo 'Hello World!'
```

Последняя строка в сценарии хорошо знакома — это простая команда `echo` со строковым аргументом. Вторая строка также знакома. Она выглядит как комментарии, которые мы видели во многих конфигурационных файлах, исследованных и отредактированных нами. Еще одна особенность комментариев, о которой пока не рассказывалось, — они могут появляться в концах строк, если им предшествует хотя бы один пробельный символ, например:

```
echo 'Hello World!' # Это тоже комментарий
```

Все, начиная с символа `#` и до конца строки, игнорируется.

То же самое верно и для командной строки:

```
[me@linuxbox ~]$ echo 'Hello World!' # Это тоже комментарий
Hello World!
```

Несмотря на бессмысленность комментариев в командной строке, их все же можно использовать.

Первая строка в сценарии смотрится несколько необычно. Она похожа на комментарий, потому что начинается с символа #, но выглядит какой-то уж слишком специальной, чтобы быть комментарием. Последовательность символов #! — это на самом деле специальная конструкция, которая называется *shebang* (произносится как «шебанг») и сообщает системе имя интерпретатора, который должен использоваться для выполнения следующего за ним текста сценария. Каждый сценарий командной оболочки должен включать это определение в первой строке.

Сохраните файл сценария с именем *hello_world*.

Разрешения на выполнение

Далее сделаем сценарий исполняемым при помощи команды chmod:

```
[me@linuxbox ~]$ ls -l hello_world
-rw-r--r-- 1 me      me      63 2018-03-07 10:10 hello_world
[me@linuxbox ~]$ chmod 755 hello_world
[me@linuxbox ~]$ ls -l hello_world
-rwxr-xr-x 1 me      me      63 2018-03-07 10:10 hello_world
```

Существует два распространенных набора разрешений для сценариев: 755 — для сценариев, которые должны быть доступны для выполнения всем, и 700 — для сценариев, которые могут выполняться только владельцами. Обратите внимание, что сценарии необходимо сделать доступными для чтения, чтобы их можно было выполнить.

Местоположение файла сценария

После установки разрешений попробуем запустить сценарий:

```
[me@linuxbox ~]$ ./hello_world
Hello World!
```

Но чтобы это сделать, необходимо добавить явный путь перед его именем. В противном случае мы получим следующее сообщение:

```
[me@linuxbox ~]$ hello_world
bash: hello_world: команда не найдена
```

В чем причина? Чем наш сценарий отличается от других программ? Как оказывается, ничем. У нас замечательный сценарий. Его проблема — местоположение. В главе 11 мы изучали переменную окружения PATH и ее влияние на то, как система

ищет выполняемые программы. Коротко напомним, что система просматривает каталоги по списку всякий раз, когда требуется найти исполняемую программу, если путь к ней не указан явно. Именно так система выполняет программу */bin/ls*, если мы вводим `ls` в командной строке. Каталог */bin* — один из каталогов, которые система просматривает автоматически. Список каталогов хранится в переменной окружения `PATH`. Она содержит список каталогов, перечисленных через двоеточие. Увидеть, что содержится в `PATH`, можно с помощью команды:

```
[me@linuxbox ~]$ echo $PATH
/home/me/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/
games
```

Как видите, это просто список каталогов. Если поместить сценарий в любой из этих каталогов, наша проблема будет решена. Обратите внимание на первый каталог в списке, */home/me/bin*. В большинстве дистрибутивов Linux в переменную `PATH` включается каталог *bin* в домашнем каталоге пользователя, чтобы дать пользователям возможность выполнять собственные программы. То есть если создать каталог *bin* и поместить сценарий в него, его можно будет запускать как любые другие программы:

```
[me@linuxbox ~]$ mkdir bin
[me@linuxbox ~]$ mv hello_world bin
[me@linuxbox ~]$ hello_world
Hello World!
```

Если каталог отсутствует в переменной `PATH`, его легко туда добавить, включив следующую строку в файл *.bashrc*:

```
export PATH=~/bin:"$PATH"
```

Это изменение будет действовать в каждом последующем сеансе работы с терминалом. Чтобы применить изменения в текущем сеансе, нужно заставить командную оболочку повторно прочитать файл *.bashrc*, например, так:

```
[me@linuxbox ~]$ . .bashrc
```

Команда «точка» (`.`) является синонимом `source`, встроенной команды, которая читает указанный файл и интерпретирует его как ввод с клавиатуры.

ПРИМЕЧАНИЕ

Ubuntu (и многие другие дистрибутивы на основе Debian) автоматически добавляет каталог *~/bin* в переменную `PATH`, если он существует в момент выполнения файла *.bashrc*. То есть если в системе Ubuntu создать каталог *~/bin* и затем выйти и войти в систему, проблема решится автоматически.

НАСТРОЙКА VIM ДЛЯ РАЗРАБОТКИ СЦЕНАРИЕВ

Текстовый редактор `vim` имеет много, очень много параметров настройки. Некоторые из них можно использовать для подготовки редактора к разработке сценариев. Следующая команда:

```
:syntax on
```

включает подсветку синтаксиса. С этой настройкой редактор будет отображать синтаксические элементы сценариев разным цветом. Это помогает выявлять некоторые виды ошибок. И конечно же, выглядит очень круто. Обратите внимание, что для работы этой настройки должна быть установлена полная версия `vim`, а редактируемый файл должен содержать строку шебанг (shebang), сообщающую, что файл является сценарием командной оболочки. Если с этой командой возникнут сложности, попробуйте настройку `:set syntax=sh`.

Следующая команда:

```
:set hlsearch
```

включает подсветку результатов поиска. Например, если выполнить поиск слова *echo* с этой настройкой, редактор выделит все вхождения искомого слова.

Следующая команда:

```
:set tabstop=4
```

определяет число колонок (знакомест), занимаемых символом табуляции. По умолчанию один символ табуляции занимает восемь знакомест. Присвоив этому параметру значение 4 (которое широко используется практикующими программистами), вам проще будет уместить длинные строки на экране.

Следующая команда:

```
:set autoindent
```

включает автоматическое оформление отступов. Этот параметр заставляет `vim` добавлять в новую строку отступ, как в строке выше. Это ускоряет ввод многих видов программных конструкций. Чтобы прекратить автоматическое выравнивание, достаточно нажать комбинацию CTRL+D.

Эти изменения можно сделать постоянными, добавив описанные команды (без начального двоеточия) в файл *~/.vimrc*.

Выбор местоположения для сценариев

Каталог *~/bin* хорошо подходит для сценария, если этот сценарий предназначен для личного использования. Сценарии, которые должны быть доступны всем

пользователям в системе, лучше размещать в традиционном месте — в каталоге */usr/local/bin*. Сценарии, предназначенные для использования системным администратором, часто помещаются в каталог */usr/local/sbin*. В большинстве случаев программное обеспечение, созданное в локальной системе, будь то сценарии или скомпилированные программы, следует помещать в иерархию каталогов */usr/local*, а не */bin* или */usr/bin*. Последние два каталога, как определено стандартом иерархии файловой системы Linux (Linux Filesystem Hierarchy Standard), предназначены только для файлов, поставляемых создателями дистрибутива Linux.

Дополнительные хитрости по оформлению

Одной из ключевых целей, стоящих перед создателями сценариев, является простота сопровождения, то есть простота, с которой сценарий может быть изменен автором или другими пользователями для удовлетворения меняющихся потребностей. Один из способов упростить сопровождение — улучшить читаемость и понятность сценария.

Длинные имена параметров

Многие команды, с которыми мы уже знакомы, поддерживают параметры с короткими и длинными именами. Например, команда `ls` имеет множество параметров, которые можно выразить в короткой и в длинной форме. Например:

```
[me@linuxbox ~]$ ls -ad
```

и

```
[me@linuxbox ~]$ ls --all --directory
```

это эквивалентные команды. Параметры с короткими именами предпочтительнее использовать в командной строке, так как они помогают уменьшить ручной ввод, но длинные имена параметров могут улучшить читаемость.

Отступы и продолжения строк

Если приходится использовать длинные команды, их читаемость можно повысить, распределяя такие команды по нескольким строкам. В главе 17 был представлен пример длинной команды find:

```
[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 -exec
chmod 0600 '{}' ';' \) -or \( -type d -not -perm 0700 -exec chmod
0700 '{}' ';' \)
```

С первой попытки понять эту команду очень сложно. В тексте сценария ее можно упростить, записав следующим образом:

```
find playground \
	\( \
		-type f \
		-not -perm 0600 \
		-exec chmod 0600 '{}' ';' \
	\) \
	-or \
	\( \
		-type d \
		-not -perm 0700 \
		-exec chmod 0700 '{}' ';' \
	\)
```

С помощью последовательностей продолжения строки (включающих обратный слеш и символ перевода строки) и отступов логику этой сложной команды удалось сделать ясной для читателя. Этот прием работает также в командной строке, однако он редко используется из-за неудобства ввода и редактирования. Одно из отличий сценариев от командной строки — возможность использования символов табуляции для оформления отступов, тогда как в командной строке это невозможно, потому что клавиша ввода табуляции активирует функцию автодополнения.

Заключение

В первой главе, посвященной сценариям, мы увидели, как писать сценарии и как упростить их запуск в своей системе. Мы также познакомились с некоторыми приемами оформления, улучшающими читаемость (и тем самым упрощающими сопровождение) сценариев. В следующих главах мы снова и снова будем возвращаться к простоте сопровождения как главному принципу создания качественных сценариев.

25 Начало проекта

В этой главе мы приступаем к созданию программы. Цель данного проекта — показать, как можно использовать разные возможности командной оболочки для создания программ и, что особенно важно, для создания *хороших* программ.

Далее мы напишем *генератор отчетов*. Он будет выводить разнообразную информацию о системе и ее состоянии в формате HTML, благодаря чему ее можно будет просматривать в веб-браузере.

Обычно создание программ выполняется в несколько этапов, на каждом из которых добавляются новые функции и возможности. По окончании первого этапа наша программа будет воспроизводить минимальную HTML-страницу без какой-либо информации. Эту информацию мы добавим на следующих этапах.

Этап первый: минимальный документ

Прежде всего, определим, как выглядит формат правильно сформированного HTML-документа. Он имеет следующий вид:

```
<html>
        <head>
            <title>Заголовок страницы</title>
        </head>
        <body>
                Тело страницы.
        </body>
</html>
```

Если ввести этот текст в текстовом редакторе и сохранить в файле с именем *foo.html*, мы сможем открыть его, введя следующий адрес URL в Firefox: *file:///home/username/foo.html*.

На первом этапе создадим программу, которая будет выводить эту разметку HTML в стандартный вывод. Написать такую программу очень просто. Откройте текстовый редактор и создайте файл с именем *~/bin/sys_info_page*:

```
[me@linuxbox ~]$ vim ~/bin/sys_info_page
```

А затем введите следующую программу:

```
#!/bin/bash
# Программа вывода страницы с информацией о системе
echo "<html>"
echo "  <head>"
echo "        <title>Page Title</title>"
echo "  </head>"
echo "  <body>"
echo "        Page body."
echo "  </body>"
echo "</html>"
```

Наша первая версия содержит строку-шебанг (shebang), комментарий (можно только приветствовать) и последовательность команд `echo`, по одной для вывода каждой строки. После сохранения файла сделайте его выполняемым и попробуйте запустить:

```
[me@linuxbox ~]$ chmod 755 ~/bin/sys_info_page
[me@linuxbox ~]$ sys_info_page
```

После запуска на экране должен появиться текст HTML-документа, потому что команды `echo` в сценарии посылают свои строки в стандартный вывод. Запустите программу снова и перенаправьте вывод программы в файл *sys_info_page.html*, чтобы затем посмотреть результат в веб-браузере:

```
[me@linuxbox ~]$ sys_info_page > sys_info_page.html
[me@linuxbox ~]$ firefox sys_info_page.html
```

Пока все идет неплохо.

Разрабатывая программы, всегда следует помнить о простоте и ясности. Сопровождение дается проще, когда программа легко читается и доступна для понимания, не говоря уже о том, что программу легче писать, когда есть возможность уменьшить объем ручного ввода. Текущая версия программы работает замечательно, но ее можно упростить. Если объединить все команды `echo` в одну, это определенно упростит в будущем добавление новых строк в вывод программы. Поэтому изменим программу, как показано ниже:

```
#!/bin/bash

# Программа вывода страницы с информацией о системе

echo "<html>
        <head>
                <title>Page Title</title>
        </head>
        <body>
                Page body.
        </body>
</HTML>"
```

Строки в кавычках могут включать символы перевода строки и, соответственно, содержать несколько строк текста. Командная оболочка будет продолжать читать текст, пока не встретит закрывающую кавычку. Это правило действует также в командной строке:

```
[me@linuxbox ~]$ echo "<html>
>         <head>
>                 <title>Page Title</title>
>         </head>
>         <body>
>                 Page body.
>         </body>
> </html>"
```

Символ > в начале каждой строки — это приглашение к вводу командной оболочки, определяемое ее переменной PS2. Оно появляется всякий раз, когда происходит ввод многострочной инструкции. Эта особенность пока малопонятна, но потом, когда мы познакомимся с многострочными программными инструкциями, ее преимущества станут очевидными.

Этап второй: добавление некоторых данных

Теперь, когда программа способна сгенерировать минимальный документ, добавим в отчет немного данных. Для этого внесите следующие изменения:

```
#!/bin/bash

# Программа вывода страницы с информацией о системе

echo "<html>
        <head>
```

```
                <title>System Information Report</title>
        </head>
        <body>
                <h1>System Information Report</h1>
        </body>
</html>"
```

Здесь добавлено название страницы и заголовок в теле отчета.

Переменные и константы

В нашем сценарии возникла проблема. Обратили внимание, что строка `System Information Report` повторяется дважды? Вообще, для такого крохотного сценария это не такая большая проблема, но представьте по-настоящему длинный сценарий, в котором эта строка повторяется много раз. Если в таком сценарии понадобится изменить название, придется внести изменения во множестве мест, а это масса ручной работы. Можно ли изменить сценарий так, чтобы строка определялась в нем только один раз? Это существенно упростило бы сопровождение сценария в будущем. Да, это возможно, например, так:

```
#!/bin/bash

# Программа вывода страницы с информацией о системе

title="System Information Report"

echo "<html>
        <head>
                <title>$title</title>
        </head>
        <body>
                <h1>$title</h1>
        </body>
</html>"
```

Создав *переменную* с именем `title` и присвоив ей значение `System Information Report`, мы воспользовались преимуществами подстановки параметров и поместили строку во множество мест.

Но как создать переменную? Просто — достаточно использовать ее. Когда командная оболочка встречает переменную, она автоматически создает ее. Этим она отличается от многих языков программирования, в которых переменные должны явно объявляться или определяться до ее использования. Командная оболочка

слишком либеральна в этом отношении, что в итоге приводит к некоторым проблемам. Например, рассмотрим следующий сценарий, выполненный в командной строке:

```
[me@linuxbox ~]$ foo="yes"
[me@linuxbox ~]$ echo $foo
yes
[me@linuxbox ~]$ echo $fool

[me@linuxbox ~]$
```

Мы сначала присвоили значение `yes` переменной `foo` и затем вывели ее значение командой `echo`. Далее, мы попробовали вновь вывести значение переменной, но допустили опечатку, указав имя `fool`, и получили пустую строку. Такой результат объясняется тем, что командная оболочка благополучно создала переменную `fool`, встретив ее, и присвоила ей пустое значение по умолчанию. Из этого примера следует, что нужно внимательно следить за правописанием! Также важно понять, что в действительности произошло в этом примере. Из предыдущего знакомства с особенностями работы механизма подстановки мы знаем, что команда

```
[me@linuxbox ~]$ echo $foo
```

подвергается действию механизма подстановки параметров, в результате чего приобретает вид

```
[me@linuxbox ~]$ echo yes
```

С другой стороны, команда

```
[me@linuxbox ~]$ echo $fool
```

превращается в

```
[me@linuxbox ~]$ echo
```

На место пустой переменной ничего не подставляется! Это может вызвать ошибку в командах, требующих наличия аргументов. Например:

```
[me@linuxbox ~]$ foo=foo.txt
[me@linuxbox ~]$ foo1=foo1.txt
[me@linuxbox ~]$ cp $foo $fool
cp: после 'foo.txt' пропущен операнд, задающий целевой файл
По команде "cp --help" можно получить дополнительную информацию.
```

Мы присвоили значения двум переменным, `foo` и `foo1`. А затем попытались выполнить команду `cp`, но допустили опечатку в имени второго аргумента. После обработки механизмом подстановки команда `cp` получила только один аргумент, хотя требует двух.

Ниже приводятся несколько правил именования переменных:

- Имена переменных могут состоять из алфавитно-цифровых символов (букв и цифр) и символов подчеркивания.
- Первый символ в имени переменной может быть только буквой или символом подчеркивания.
- Присутствие пробелов и знаков препинания в именах переменных не допускается.

Название *переменная* подразумевает значение, которое может изменяться, и во многих приложениях переменные именно так и используются. Однако переменная `title` в нашем приложении используется как *константа*. Константа, так же как переменная, имеет имя и содержит значение. Отличие лишь в том, что значение константы не изменяется. В приложении, осуществляющем геометрические расчеты, можно определить константу `PI` со значением `3.1415`, вместо того, чтобы использовать это число по всей программе. Командная оболочка не различает константы и переменные; эти термины используются в основном для удобства программиста. Типичное соглашение — использовать буквы верхнего регистра для обозначения констант и буквы нижнего регистра для истинных переменных. Давайте изменим сценарий, приведя его в соответствие с этим соглашением:

```
#!/bin/bash

# Программа вывода страницы с информацией о системе

TITLE="System Information Report For $HOSTNAME"

echo "<html>
        <head>
                <title>$TITLE</title>
        </head>
        <body>
                <h1>$TITLE</h1>
        </body>
</html>"
```

Попутно мы дополнили название, добавив в конец значение переменной командной оболочки `HOSTNAME`. Это — сетевое имя машины.

ПРИМЕЧАНИЕ

В действительности командная оболочка имеет механизм, гарантирующий неизменяемость констант, в виде встроенной команды `declare` с параметром `-r` (read-only — только для чтения). Если переменной `TITLE` присвоить значение, как показано ниже:

```
declare -r TITLE="Page Title"
```

командная оболочка не допустит повторного присваивания значения переменной `TITLE`. Этот механизм редко используется на практике, но он имеется и его можно применять в особенно строгих сценариях.

Присваивание значений переменным и константам

Мы подошли к моменту, когда наше знание особенностей работы механизма подстановки начинает приносить свои плоды. Как мы видели, присваивание значений переменным производится так:

переменная=значение

где *переменная* — это имя переменной, а *значение* — строка. В отличие от некоторых других языков программирования, командная оболочка не заботится о типах значений, присваиваемых переменным; она все значения интерпретирует как строки. Существует возможность заставить командную оболочку ограничить круг присваиваемых значений целыми числами, задействовав команду `declare` с параметром `-i`, но, как и объявление переменных, доступных только для чтения, эта возможность редко используется на практике.

Обратите внимание на отсутствие пробелов в операторе присваивания между именем переменной, знаком «равно» и значением. А из чего может состоять значение? Из всего что угодно, что можно развернуть в строку.

```
a=z                 # Присвоит переменной a строку "z".
b="a string"        # Внутренние пробелы должны находиться в кавычках.
c="a string and $b" # При присваивании допускается выполнять подстановку,
                    # например, значений других переменных.
d=$(ls -l foo.txt)  # Результат выполнения команды.
e=$((5 * 7))        # Подстановка результата арифметического выражения.
f="\t\ta string\n"  # Экранированные последовательности, такие как
                    # символы табуляции и перевода строки.
```

В одной строке можно выполнить присваивание сразу нескольким переменным:

```
a=5 b="a string"
```

При использовании подстановки имена переменных можно заключать в необязательные фигурные скобки {}. Это пригодится в том случае, когда имя переменной становится неоднозначным в окружающем контексте. В следующем примере выполняется попытка переименовать файл *myfile* в *myfile1* с использованием переменной:

```
[me@linuxbox ~]$ filename="myfile"
[me@linuxbox ~]$ touch $filename
[me@linuxbox ~]$ mv $filename $filename1
mv: после 'myfile' пропущен операнд, задающий целевой файл
По команде "mv --help" можно получить дополнительную информацию.
```

Эта попытка не увенчалась успехом, потому что командная оболочка интерпретировала второй аргумент команды `mv` как имя новой (и пустой) переменной. Ниже показано, как решается подобная проблема:

```
[me@linuxbox ~]$ mv $filename ${filename}1
```

Добавив фигурные скобки, мы гарантировали, что командная оболочка не будет интерпретировать последний символ `1` как часть имени переменной.

ПРИМЕЧАНИЕ

Выполняя подстановку, имена переменных и команды рекомендуется заключать в двойные кавычки, чтобы исключить разбиение строк на слова оболочкой. Особенно важно использовать кавычки, когда переменная может содержать имя файла.

Воспользуемся этой возможностью, чтобы добавить в отчет дополнительные данные, а именно дату и время составления отчета, а также имя пользователя, составившего отчет:

```
#!/bin/bash

# Программа вывода страницы с информацией о системе

TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

echo "<html>
        <head>
                <title>$TITLE</title>
        </head>
```

```
        <body>
            <h1>$TITLE</h1>
            <p>$TIME_STAMP</p>
        </body>
</html>"
```

Встроенные документы

Мы рассмотрели два разных метода вывода текста, и оба используют команду echo. Однако существует еще один, третий метод, который называется *встроенным документом* (here document), или *встроенным сценарием* (here script). Встроенный документ — это дополнительная форма перенаправления ввода/вывода, которая передает текст, встроенный в сценарий, на стандартный ввод команды. Действует это перенаправление так:

```
команда << индикатор
текст
индикатор
```

где *команда* — это имя команды, принимающей указанный текст через стандартный ввод, а *индикатор* — это строка, отмечающая конец встроенного текста. Изменим сценарий, задействовав в нем встроенный документ:

```
#!/bin/bash

# Программа вывода страницы с информацией о системе

TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

cat << _EOF_
<html>
        <head>
                <title>$TITLE</title>
        </head>
        <body>
            <h1>$TITLE</h1>
            <p>$TIME_STAMP</p>
        </body>
</html>
_EOF_
```

Теперь вместо команды echo в сценарии используются команда cat и встроенный документ. На роль индикатора была выбрана строка _EOF_ (означает

end-of-file — конец файла, распространенное соглашение), и она отмечает конец встроенного текста. Обратите внимание, что строка-индикатор должна находиться в отдельной строке, одна, и за ней не должно следовать никаких пробелов.

Но какие преимущества дало использование встроенного документа здесь? Практически никаких, кроме того, что кавычки внутри встроенных документов теряют свое специальное значение для командной оболочки. Ниже приводится пример использования встроенного документа в командной строке:

```
[me@linuxbox ~]$ foo="some text"
[me@linuxbox ~]$ cat << _EOF_
> $foo
> "$foo"
> '$foo'
> \$foo
> _EOF_
some text
"some text"
'some text'
$foo
```

Как видите, командная оболочка не обращает никакого внимания на кавычки. Она интерпретирует их как обычные символы. Благодаря этому мы свободно вставляем кавычки во встроенные документы. Этим обстоятельством можно воспользоваться при разработке программ составления отчетов.

Встроенные документы можно использовать с любыми командами, принимающими данные со стандартного ввода. В следующем примере встроенный документ используется для передачи последовательности команд программе `ftp`, чтобы загрузить файл с удаленного FTP-сервера:

```
#!/bin/bash

# Сценарий загрузки файла через FTP

FTP_SERVER=ftp.nl.debian.org
FTP_PATH=/debian/dists/stretch/main/installer-amd64/current/images/cdrom
REMOTE_FILE=debian-cd_info.tar.gz

ftp -n << _EOF_
open $FTP_SERVER
user anonymous me@linuxbox
cd $FTP_PATH
hash
get $REMOTE_FILE
```

```
bye
_EOF_
ls -l $REMOTE_FILE
```

Если заменить оператор перенаправления << на <<-, командная оболочка будет игнорировать начальные символы табуляции во встроенном документе. Благодаря этому во встроенный документ можно добавить отступы для большей удобочитаемости:

```
#!/bin/bash

# Сценарий загрузки файла через FTP

FTP_SERVER=ftp.nl.debian.org
FTP_PATH=/debian/dists/stretch/main/installer-amd64/current/images/cdrom
REMOTE_FILE=debian-cd_info.tar.gz

ftp -n <<- _EOF_
	open $FTP_SERVER
	user anonymous me@linuxbox
	cd $FTP_PATH
	hash
	get $REMOTE_FILE
	bye
	_EOF_
ls -l $REMOTE_FILE
```

Однако пользоваться этой особенностью не всегда удобно, потому что для оформления отступов многие текстовые редакторы (и сами программисты) предпочитают использовать символы пробела вместо символов табуляции.

Заключение

В этой главе мы приступили к разработке проекта, при помощи которого пройдем через все этапы создания сценария. Мы познакомились с переменными и константами и особенностями их использования. Они чаще других программных компонентов применяются для подстановки. Мы также увидели, как организовать вывод информации в сценарии, и познакомились с разными методами встраивания блоков текста.

26 Проектирование сверху вниз

С увеличением размеров и сложности программ их становится все труднее проектировать, программировать и сопровождать. Практически к любому сложному проекту с успехом можно применить методологию деления больших и сложных задач на более мелкие и простые.

Представьте, что нам нужно описать типичную повседневную задачу — сходить в магазин и купить продукты — для пришельца с Марса. Весь процесс можно разбить на следующую последовательность шагов:

1. Сесть в машину.
2. Доехать до магазина.
3. Припарковать машину.
4. Войти в магазин.
5. Купить продукты.
6. Вернуться в машину.
7. Доехать до дома.
8. Припарковать машину.
9. Войти в дом.

Однако инопланетянину с Марса почти наверняка потребуется больше деталей. Задачу «Припарковать машину» мы могли бы разбить на еще более мелкие шаги.

1. Найти место на парковке.
2. Поставить машину на это место.
3. Выключить двигатель.
4. Поставить на ручной тормоз.
5. Выйти из машины.
6. Запереть машину.

Подзадачу «Выключить двигатель» можно разбить на еще более мелкие шаги, например «Выключить зажигание», «Вынуть ключ зажигания» и так далее, пока все шаги посещения магазина не будут определены во всех деталях.

Подобный процесс идентификации высокоуровневых шагов и проработку все более мелких деталей этих шагов называют *проектированием сверху вниз*. Этот прием позволяет разбивать большие, сложные задачи на множество мелких и простых задач. Проектирование сверху вниз часто используется в разработке программного обеспечения и хорошо подходит для программирования на языке командной оболочки.

В этой главе воспользуемся приемом проектирования сверху вниз для дальнейшей разработки сценария генератора отчетов.

Функции командной оболочки

В настоящий момент наш сценарий генерирует документ HTML, выполняя следующие шаги:

1. Открыть страницу.
2. Открыть заголовок страницы.
3. Установить название страницы.
4. Закрыть заголовок страницы.
5. Открыть тело страницы.
6. Вывести заголовок на странице.
7. Вывести текущее время.
8. Закрыть тело страницы.
9. Закрыть страницу.

На следующем этапе разработки мы добавим несколько задач между шагами 7 и 8:

- **Продолжительность непрерывной работы системы и степень ее загруженности** — это интервал времени, прошедшего с момента последней загрузки системы, и среднее число задач, выполняемых процессором в настоящее время для нескольких отрезков времени.
- **Дисковое пространство** — информация об использовании дискового пространства на системных устройствах хранения.
- **Объем домашних каталогов** — объем дискового пространства, занятого каждым пользователем.

Если бы у нас были команды, решающие перечисленные задачи, мы бы просто добавили их в сценарий, воспользовавшись механизмом подстановки результатов команд:

```
#!/bin/bash

# Программа вывода страницы с информацией о системе

TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

cat << _EOF_
<html>
        <head>
                <title>$TITLE</title>
        </head>
        <body>
                <h1>$TITLE</h1>
                <p>$TIME_STAMP</p>
                $(report_uptime)
                $(report_disk_space)
                $(report_home_space)
        </body>
</html>
_EOF_
```

Создать такие команды можно двумя способами: написать три отдельных сценария и поместить их в каталог, входящий в список PATH, или встроить эти сценарии в программу в виде *функций командной оболочки*. Как уже отмечалось ранее, функции — это «мини-сценарии», находящиеся внутри другого сценария, которые работают как автономные программы. Функции имеют две синтаксические формы. Первая выглядит так:

```
function имя {
    команды
    return
}
```

Вторая проще (и часто предпочтительнее):

```
имя () {
    команды
    return
}
```

где *имя* — это имя функции, а ***команды*** — последовательность команд внутри функции. Обе формы эквивалентны и могут использоваться одна вместо другой. Ниже приводится сценарий, демонстрирующий использование функций командной оболочки:

```
1     #!/bin/bash
2
3     # Демонстрация функций командной оболочки
4
5     function step2 {
6         echo "Step 2"
7         return
8     }
9
10    # Здесь начинается основная программа
11
12    echo "Step 1"
13    step2
14    echo "Step 3"
```

Когда командная оболочка читает сценарий, она пропускает строки с 1-й по 11-ю, так как они содержат комментарии и определение функции. Выполнение начинается со строки 12 с командой `echo`. Строка 13 *вызывает* функцию `step2`, и командная оболочка выполняет функцию как любую другую команду. Управление передается в строку 6, и выполняется вторая команда `echo`. Следующей выполняется строка 7. Команда `return` в этой строке завершает выполнение функции и возвращает управление в строку, следующую за вызовом функции (строка 14). После этого выполняется заключительная команда `echo`. Обратите внимание: чтобы вызовы функций интерпретировались не как имена внешних программ, а действительно как вызовы функций, эти функции должны быть определены в сценарии до их вызова.

Добавим в наш сценарий минимальные определения функций:

```
#!/bin/bash

# Программа вывода страницы с информацией о системе

TITLE="System Information Report For $HOSTNAME"
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

report_uptime () {
      return
}
```

```
report_disk_space () {
      return
}
report_home_space () {
      return
}

cat << _EOF_
<html>
        <head>
                <title>$TITLE</title>
        </head>
        <body>
                <h1>$TITLE</h1>
                <p>$TIME_STAMP</p>
                $(report_uptime)
                $(report_disk_space)
                $(report_home_space)
        </body>
</html>
_EOF_
```

Имена функций подчиняются тем же правилам, что и имена переменных. Функция должна содержать хотя бы одну команду. Команда `return` (которая является необязательной) помогает удовлетворить это требование.

Локальные переменные

В сценариях, что нам доводилось писать до сих пор, все переменные (включая константы) были *глобальными*. Глобальные переменные существуют и доступны в любой точке программы. В некоторых случаях это безусловно полезное свойство осложняет использование функций. Внутри функций иногда желательно использовать *локальные переменные*. Локальные переменные доступны только внутри функции, в которой они определены, и прекращают свое существование по завершении выполнения функции.

Поддержка локальных переменных позволяет программисту использовать переменные с именами, которые уже определены в сценарии, глобально или в других функциях, не беспокоясь о возможных конфликтах имен.

Следующий пример сценария демонстрирует, как определяются и используются локальные переменные:

```
#!/bin/bash

# local-vars: сценарий, демонстрирующий локальные переменные

foo=0    # глобальная переменная foo

funct_1 () {

      local foo    # переменная foo, локальная для funct_1

      foo=1
      echo "funct_1: foo = $foo"
}

funct_2 () {

      local foo    # переменная foo, локальная для funct_2

      foo=2
      echo "funct_2: foo = $foo"
}

echo "global:  foo = $foo"
funct_1
echo "global:  foo = $foo"
funct_2
echo "global:  foo = $foo"
```

Как видите, локальные переменные объявляются добавлением слова `local` перед именем переменной. В результате создается переменная, локальная по отношению к функции, в которой она определена. Когда выполнение выйдет за пределы функции, переменная перестанет существовать. Если запустить этот сценарий, он выведет следующее:

```
[me@linuxbox ~]$ local-vars
global:  foo = 0
funct_1: foo = 1
global:  foo = 0
funct_2: foo = 2
global:  foo = 0
```

Этот пример показывает, что присваивание значений локальной переменной `foo` внутри обеих функций не оказывает влияния на значение переменной `foo`, объявленной за пределами функций.

Эта особенность позволяет писать функции, сохраняя их независимость друг от друга и от сценария, в котором они определяются. Это очень ценное качество, оно предотвращает взаимовлияние разных частей программы друг на друга, а кроме того, помогает писать переносимые функции, то есть функции, которые можно скопировать из одного сценария в другой.

Постоянное опробование сценария

В процессе разработки программ необходимо постоянно проверять их работоспособность. Запуская и тестируя программы как можно чаще, мы сможем выявить ошибки на самых ранних этапах разработки. Это существенно упрощает задачу отладки. Например, если после внесения небольших изменений и очередного запуска программы обнаружится ошибка, источник проблемы почти наверняка будет находиться в последних изменениях. Добавив пустые функции, которые на языке программистов называются *заглушками*, мы смогли проверить работоспособность программы на ранней стадии. Создавая заглушку, неплохо было бы включить в нее что-то, что давало бы обратную связь, позволяющую программисту оценить ход выполнения. Если сейчас взглянуть на вывод нашего сценария, можно заметить несколько пустых строк, следующих за строкой с текущим временем, но мы пока не уверены в причинах их появления.

```
[me@linuxbox ~]$ sys_info_page
<html>
        <head>
                <title>System Information Report For twin2</title>
        </head>
        <body>
                <h1>System Information Report For linuxbox</h1>
                <pP>Generated 03/19/2012 04:02:10 PM EDT, by me</p>



        </body>
</html>
```

Изменим функции, добавив в них сообщения для обратной связи:

```
report_uptime () {
        echo "Function report_uptime executed."
        return
}

report_disk_space () {
```

```
        echo "Function report_disk_space executed."
        return
}

report_home_space () {
        echo "Function report_home_space executed."
        return
}
```

И запустим сценарий еще раз:

```
[me@linuxbox ~]$ sys_info_page
<html>
        <head>
                <title>System Information Report For linuxbox</title>
        </head>
        <body>
                <h1>System Information Report For linuxbox</h1>
                <p>Generated 03/20/2012 05:17:26 AM EDT, by me</p>
                Function report_uptime executed.
                Function report_disk_space executed.
                Function report_home_space executed.
        </body>
</html>
```

Теперь можно с уверенностью сказать, что наши три функции выполняются как надо.

Теперь, когда каркас функций готов и работает, самое время добавить в них некий код. Сначала займемся функцией `report_uptime`:

```
report_uptime () {
        cat <<- _EOF_
                <h2>System Uptime</h2>
                <pre>$(uptime)</pre>
                _EOF_
        return
}
```

Она выглядит очень просто. Мы использовали встроенный документ для вывода заголовка раздела и результатов выполнения команды `uptime`, заключив их в теги `<pre>`, чтобы сохранить формат вывода команды. Функция `report_disk_space` выглядит аналогично:

```
report_disk_space () {
        cat <<- _EOF_
```

```
                <h2>Disk Space Utilization</h2>
                <pre>$(df -h)</pre>
                _EOF_
        return
}
```

Она получает информацию о дисковом пространстве с помощью команды `df -h`. Наконец, определим функцию `report_home_space`:

```
report_home_space () {
        cat <<- _EOF_
                <h2>Home Space Utilization</h2>
                <pre>$(du -sh /home/*)</pre>
                _EOF_
        return
}
```

Для решения поставленной задачи мы использовали команду `du` с параметрами `-sh`. Однако это не полное решение задачи. Даже притом, что его можно использовать в некоторых системах (например, в Ubuntu), кое-где оно работать не будет. Причина в том, что во многих системах для домашних каталогов выбираются разрешения, не позволяющие читать их содержимое другим пользователям, что является вполне разумной мерой предосторожности. В этих системах функция `report_home_space` в том виде, в каком она написана здесь, будет работать, только если запустить сценарий с правами суперпользователя. Лучшее, что можно сделать в такой ситуации, — корректировать поведение сценария в соответствии с привилегиями пользователя, запустившего его. Мы займемся этим в главе 27.

ФУНКЦИИ КОМАНДНОЙ ОБОЛОЧКИ В ФАЙЛЕ .BASHRC

Функции командной оболочки могут служить прекрасной заменой псевдонимам и в действительности считаются предпочтительным способом определения небольших команд для личного использования. Возможности псевдонимов весьма ограниченны в отношении использования некоторых видов команд и особенностей командной оболочки, тогда как функции позволяют все, что можно выразить в виде сценария. Например, если вам понравилась функция `report_disk_space`, созданная нами для нашего сценария, вы можете создать похожую функцию с именем `ds` в своем файле *.bashrc*:

```
ds () {
      echo "Disk Space Utilization For $HOSTNAME"
      df -h
}
```

Заключение

В этой главе мы познакомились с широко применяемым методом проектирования программ сверху вниз и увидели, как можно поэтапно развивать функции командной оболочки. Мы также научились при помощи локальных переменных делать функции независимыми от других функций и программ, в которых они находятся. Функции можно делать переносимыми и пригодными для *повторного использования* во множестве программ, что поможет сэкономить массу времени.

27 Управление потоком выполнения: ветвление при помощи if

В предыдущей главе мы столкнулись с проблемой. Как помочь сценарию адаптировать свое поведение в зависимости от привилегий пользователя, запустившего его? Для решения проблемы нам необходим некий способ «изменить направление» выполнения сценария, опираясь на результаты проверки. Выражаясь языком программистов, нам нужен способ, обеспечивающий *ветвление* программы.

Рассмотрим простой пример логики, выраженный в псевдокоде, имитирующем язык компьютеров, но понятном человеку:

```
x = 5
Если x = 5, тогда:
    Сказать "x равно 5".
Иначе:
    Сказать "x не равно 5".
```

Это — пример ветвления. Если условие «x = 5?» верно, выполняется строка: «Сказать ‘x равно 5’». Иначе выполняется строка: «Сказать ‘x не равно 5’».

Инструкция if

В сценариях на языке командной оболочки описанную выше логику можно реализовать так:

```
x=5

if [ $x = 5 ]; then
     echo "x equals 5."
```

```
else
     echo "x does not equal 5."
fi
```

То же самое можно выполнить непосредственно в командной строке, получается немного короче:

```
[me@linuxbox ~]$ x=5
[me@linuxbox ~]$ if [ $x = 5 ]; then echo "equals 5"; else echo
"does not equal 5"; fi
equals 5
[me@linuxbox ~]$ x=0
[me@linuxbox ~]$ if [ $x = 5 ]; then echo "equals 5"; else echo
"does not equal 5"; fi
does not equal 5
```

В этом примере мы выполнили команду дважды. Первый раз со значением 5 в переменной `x`, что привело к выводу строки `equals 5`, и второй раз со значением 0 в переменной `x`, что привело к выводу строки `not equal 5`.

Инструкция `if` имеет следующий синтаксис:

```
if команды; then
      команды
[elif команды; then
      команды...]
[else
      команды]
fi
```

где ***команды*** — это список команд. На первый взгляд такой синтаксис выглядит запутанным. Но прежде чем прояснить его, посмотрим, как командная оболочка определяет, успешно или нет выполнена команда.

Код завершения

Команды (включая сценарии и функции, написанные нашими собственными руками) по завершении работы возвращают системе значение, которое называют *кодом завершения* (exit status). Это значение — целое число в диапазоне от 0 до 255 — сообщает об успешном или неуспешном завершении команды. По соглашениям значение 0 служит признаком успешного завершения, а любое другое — неуспешного. Командная оболочка поддерживает переменную, посредством которой можно определить код завершения. Например:

```
[me@linuxbox ~]$ ls -d /usr/bin
/usr/bin
[me@linuxbox ~]$ echo $?
0
[me@linuxbox ~]$ ls -d /bin/usr
ls: cannot access /bin/usr: No such file or directory
[me@linuxbox ~]$ echo $?
2
```

В этом примере мы дважды выполнили команду `ls`. В первый раз команда выполнилась благополучно. Если вывести значение переменной `$?`, можно увидеть, что оно равно 0. Во второй раз команда `ls` сообщила об ошибке, а переменная `$?` содержала значение 2, указывающее, что команда столкнулась с ошибкой. Одни команды используют разные коды завершения, чтобы сообщить о характере ошибки, тогда как другие, столкнувшись с любой ошибкой, просто возвращают значение 1. Страницы справочного руководства часто включают раздел с заголовком «Exit Status» («Коды завершения»), описывающий возвращаемые коды. Однако 0 всегда служит признаком успешного выполнения.

Командной оболочкой поддерживаются две чрезвычайно простые встроенные команды, которые просто завершаются с кодом 0 или 1. Команда `true` всегда завершается с признаком успеха, а команда `false` — всегда с признаком ошибки:

```
[me@linuxbox ~]$ true
[me@linuxbox ~]$ echo $?
0
[me@linuxbox ~]$ false
[me@linuxbox ~]$ echo $?
1
```

Эти команды можно использовать для исследования особенностей работы инструкции `if`. Инструкция `if` в действительности просто оценивает код завершения команды:

```
[me@linuxbox ~]$ if true; then echo "It's true."; fi
It's true.
[me@linuxbox ~]$ if false; then echo "It's true."; fi
[me@linuxbox ~]$
```

Команда `echo "It's true."` выполняется, только если команда, следующая за `if`, завершается успешно, и не выполняется, если команда, следующая за `if`, завершается с признаком ошибки. Если за `if` следует список команд, успешность выполнения всего списка определяется по последней команде:

```
[me@linuxbox ~]$ if false; true; then echo "It's true."; fi
It's true.
```

```
[me@linuxbox ~]$ if true; false; then echo "It's true."; fi
[me@linuxbox ~]$
```

Команда test

Вне всяких сомнений, чаще всего с инструкцией `if` используется команда `test`. Команда `test` может выполнять различные проверки и сравнения. Она имеет две эквивалентные формы:

```
test выражение
```

и более популярную

```
[ выражение ]
```

где ***выражение*** возвращает истинное (true) или ложное (false) значение. Команда `test` возвращает код завершения 0, если выражение истинно, и код завершения 1, если выражение ложно.

Выражения для проверки файлов

В табл. 27.1 перечислены выражения, используемые для проверки файлов.

Таблица 27.1. Выражения для проверки файлов

Выражение	Истинно, если
`файл1 -ef файл2`	файл1 и файл2 имеют одно и то же число индексного узла (inode; то есть два имени принадлежат жестким ссылкам, ссылающимся на один и тот же файл)
`файл1 -nt файл2`	файл1 новее файла файл2
`файл1 -ot файл2`	файл1 старше файла файл2
`-b файл`	файл существует и является специальным файлом блочного устройства
`-c файл`	файл существует и является специальным файлом символьного устройства
`-d файл`	файл существует и является каталогом
`-e файл`	файл существует

Таблица 27.1 (*окончание*)

Выражение	Истинно, если
`-f файл`	файл существует и является обычным файлом
`-g файл`	файл существует и имеет атрибут set-group-ID (бит *setgid*)
`-G файл`	файл существует и принадлежит действующей группе
`-k файл`	файл существует и имеет атрибут «*sticky bit*»
`-L файл`	файл существует и является символической ссылкой
`-O файл`	файл существует и принадлежит действующему пользователю
`-p файл`	файл существует и является именованным каналом
`-r файл`	файл существует и доступен для чтения (имеет разрешение на чтение для действующего пользователя)
`-s файл`	файл существует и имеет размер больше нуля
`-S файл`	файл существует и является сетевым сокетом
`-t дескриптор_файла`	`дескриптор_файла` представляет файл, подключенный к терминалу. Это выражение можно использовать для проверки стандартных потоков ввода/вывода/ошибок
`-u файл`	файл существует и имеет атрибут *setuid*
`-w файл`	файл существует и доступен для записи (имеет разрешение на запись для действующего пользователя)
`-x файл`	файл существует и доступен для выполнения (имеет разрешение на выполнение для действующего пользователя)

Следующий сценарий демонстрирует применение некоторых выражений с файлами:

```
#!/bin/bash

# test-file: проверка файла

FILE=~/.bashrc

if [ -e "$FILE" ]; then
        if [ -f "$FILE" ]; then
                echo "$FILE is a regular file."
```

```
        fi
        if [ -d "$FILE" ]; then
                echo "$FILE is a directory."
        fi
        if [ -r "$FILE" ]; then
                echo "$FILE is readable."
        fi
        if [ -w "$FILE" ]; then
                echo "$FILE is writable."
        fi
        if [ -x "$FILE" ]; then
                echo "$FILE is executable/searchable."
        fi
else
        echo "$FILE does not exist"
        exit 1
fi
exit
```

Сценарий проверяет файл, имя которого присвоено константе `FILE`, и выводит результат. Этот сценарий имеет две интересные особенности, на которые следует обратить внимание. Во-первых, отметьте, что параметр `$FILE` внутри выражений заключен в кавычки. Это не является обязательным требованием, но защищает от случаев, когда параметр пуст. Если механизм подстановки заменит `$FILE` пустым значением, это приведет к ошибке (операторы в этом случае будут интерпретироваться как непустые строки, а не как операторы). Использование кавычек гарантирует, что за оператором всегда будет следовать строка, даже если она пустая. Во-вторых, обратите внимание на команду `exit` (в конце сценария). Команда `exit` принимает единственный необязательный аргумент, определяющий код возврата сценария. В отсутствие аргумента `exit` вернет значение по умолчанию 0. Такое использование `exit` позволит сценарию сообщить об ошибке, если в `$FILE` содержится имя несуществующего файла. Команда `exit` в самом конце сценария добавлена исключительно для формальности. Когда командная оболочка достигает конца сценария (то есть конца файла), она в любом случае завершает выполнение сценария с кодом завершения 0.

Функции командной оболочки тоже могут возвращать свой код завершения, передавая целочисленный аргумент команде `return`. Чтобы преобразовать сценарий, приведенный выше, в функцию для использования в больших программах, нужно заменить команды `exit` инструкциями `return`:

```
test_file () {

        # test-file: проверка файла
```

```
	FILE=~/.bashrc

	if [ -e "$FILE" ]; then
		if [ -f "$FILE" ]; then
			echo "$FILE is a regular file."
		fi
		if [ -d "$FILE" ]; then
			echo "$FILE is a directory."
		fi
		if [ -r "$FILE" ]; then
			echo "$FILE is readable."
		fi
		if [ -w "$FILE" ]; then
			echo "$FILE is writable."
		fi
		if [ -x "$FILE" ]; then
			echo "$FILE is executable/searchable."
		fi
	else
		echo "$FILE does not exist"
		return 1
	fi
}
```

Выражения для проверки строк

В табл. 27.2 перечислены выражения, используемые для проверки строк.

Таблица 27.2. Выражения для проверки строк

Выражение	Истинно, если
`строка`	`строка` не пустая
`-n строка`	Длина строки больше нуля
`-z строка`	Длина строки равна нулю
`строка1 = строка2` `строка1 == строка2`	`строка1` и `строка2` равны. Допускается использовать один или два знака «равно», но предпочтительнее два
`строка1 != строка2`	`строка1` и `строка2` не равны
`строка1 > строка2`	`строка1` больше, чем `строка2`, в смысле алфавитной сортировки
`строка1 < строка2`	`строка1` меньше, чем `строка2`, в смысле алфавитной сортировки

ВНИМАНИЕ

При использовании с командой `test` операторы > и < необходимо заключать в кавычки (или экранировать символом «обратный слеш»). Если этого не сделать, они будут интерпретироваться командной оболочкой как операторы перенаправления, что может привести к плачевным результатам. Обратите также внимание: в документации к командной оболочке `bash` утверждается, что порядок сортировки соответствует порядку алфавитной сортировки, определяемому текущими региональными настройками, но в действительности это не так. В версиях `bash`, вплоть до 4.0, используется порядок сортировки ASCII (POSIX). Эта проблема была исправлена в версии 4.1.

Следующий сценарий демонстрирует применение выражений для проверки строк:

```
#!/bin/bash

# test-string: проверка значения строки

ANSWER=maybe

if [ -z "$ANSWER" ]; then
      echo "There is no answer." >&2
      exit 1
fi

if [ "$ANSWER" == "yes" ]; then
      echo "The answer is YES."
elif [ "$ANSWER" == "no" ]; then
      echo "The answer is NO."
elif [ "$ANSWER" == "maybe" ]; then
      echo "The answer is MAYBE."
else
      echo "The answer is UNKNOWN."
fi
```

В этом сценарии определяется константа `ANSWER`. Сначала сценарий проверяет, не является ли строка пустой. Если строка пустая, сценарий завершается с кодом 1. Обратите внимание на оператор перенаправления в команде `echo`. Он перенаправляет сообщение об ошибке «There is no answer» («Нет ответа») в стандартный вывод ошибок как «наиболее подходящий» для сообщений об ошибках. Если строка не пустая, сценарий сравнивает ее значение со строками «yes», «no» или «maybe». Проверки выполняются с использованием инструкции `elif`, которая является

краткой формой записи для `else if`. Инструкция `elif` позволяет конструировать более сложные логические проверки.

Выражения для проверки целых чисел

В табл. 27.3 перечислены выражения, используемые для проверки целых чисел.

Таблица 27.3. Выражения для проверки целых чисел

Выражение	Истинно, если
`число1 -eq число2`	число1 и число2 равны
`число1 -ne число2`	число1 и число2 не равны
`число1 -le число2`	число1 меньше или равно числу2
`число1 -lt число2`	число1 меньше, чем число2
`число1 -ge число2`	число1 больше или равно числу2
`число1 -gt число2`	число1 больше, чем число2

Следующий сценарий демонстрирует их применение:

```
#!/bin/bash

# test-integer: проверка целочисленного значения.

INT=-5

if [ -z "$INT" ]; then
      echo "INT is empty." >&2
      exit 1
fi

if [ $INT -eq 0 ]; then
      echo "INT is zero."
else
      if [ $INT -lt 0 ]; then
            echo "INT is negative."
      else
            echo "INT is positive."
      fi
      if [ $((INT % 2)) -eq 0 ]; then
            echo "INT is even."
```

```
        else
            echo "INT is odd."
        fi
fi
```

Обратите внимание на то, как сценарий определяет четность (even) или нечетность (odd) целого числа. Он возвращает остаток от деления числа на 2, по значению которого можно судить о четности или нечетности числа.

Более современная версия команды test

Последние версии `bash` реализуют составную команду, которая действует как улучшенная замена для команды `test`. Она имеет следующий синтаксис:

```
[[ выражение ]]
```

где ***выражение*** возвращает истинное (true) или ложное (false) значение. Команда `[[ ]]` очень похожа на команду `test` (она поддерживает те же выражения), но добавляет новое выражение для проверки строк:

```
строка1 =~ регулярное_выражение
```

возвращающее истинное значение, если `строка1` соответствует расширенному регулярному выражению. Это открывает широкие перспективы для решения таких задач, как проверка корректности данных. Предыдущий сценарий, демонстрирующий применение выражений проверки целых чисел, может завершиться с ошибкой, если константе `INT` присвоить любое значение, не являющееся целым числом. Для надежности сценарию необходима возможность убедиться, что константа действительно содержит целое число. Используя `[[ ]]` с оператором проверки строки `=~`, мы усовершенствуем его, как показано ниже:

```
#!/bin/bash

# test-integer2: проверка целочисленного значения.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
        if [ $INT -eq 0 ]; then
            echo "INT is zero."
        else
            if [ $INT -lt 0 ]; then
                echo "INT is negative."
```

```
            else
                  echo "INT is positive."
            fi
            if [ $((INT % 2)) -eq 0 ]; then
                  echo "INT is even."
            else
                  echo "INT is odd."
            fi
      fi
else
      echo "INT is not an integer." >&2
      exit 1
fi
```

Применив регулярное выражение, мы смогли ограничить круг проверяемых значений в константе INT только строками, начинающимися с необязательного знака «минус», за которым следует одна или несколько цифр. Это выражение также устраняет вероятность появления пустых значений.

Еще одна дополнительная особенность [[ ]]: оператор == поддерживает сопоставление с шаблоном по аналогии с механизмом подстановки путей. Например:

```
[me@linuxbox ~]$ FILE=foo.bar
[me@linuxbox ~]$ if [[ $FILE == foo.* ]]; then
> echo "$FILE matches pattern 'foo.*'"
> fi
foo.bar matches pattern 'foo.*'
```

Она превращает [[ ]] в удобный инструмент проверки имен файлов и путей.

(( )) — для проверки целых чисел

В дополнение к составной команде [[ ]] bash поддерживает также составную команду (( )), которую удобно использовать для работы с целыми числами. Она поддерживает полное множество арифметических операторов, о которых подробно рассказывается в главе 34.

Команда (( )) применяется для проверки истинности арифметических выражений. Арифметическое выражение считается истинным, если его результат отличается от нуля.

```
[me@linuxbox ~]$ if ((1)); then echo "It is true."; fi
It is true.
[me@linuxbox ~]$ if ((0)); then echo "It is true."; fi
[me@linuxbox ~]$
```

Применив `(( ))`, можно немного упростить сценарий `test-integer2`, как показано ниже:

```
#!/bin/bash

# test-integer2a: проверка целочисленного значения.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
      if ((INT == 0)); then
            echo "INT is zero."
      else
            if ((INT < 0)); then
                  echo "INT is negative."
            else
                  echo "INT is positive."
            fi
            if (( ((INT % 2)) == 0)); then
                  echo "INT is even."
            else
                  echo "INT is odd."
            fi
      fi
else
      echo "INT is not an integer." >&2
      exit 1
fi
```

Обратите внимание, что здесь мы использовали знак «меньше», а равенство проверяется с помощью оператора `==`. Такой синтаксис выглядит более естественным при работе с целыми числами. Отметьте также, что составная команда `(( ))` является частью синтаксиса командной оболочки, а не обычной командой, может применяться только к целым числам, распознает переменные по именам и не требует выполнять подстановку. Подробнее команду `(( ))` и связанную с ней подстановку значения арифметического выражения мы обсудим в главе 34.

Объединение выражений

Для более сложных вычислений существует возможность объединения выражений. Объединяются выражения с помощью логических операторов. Мы уже встречались с ними в главе 17, когда изучали команду `find`. Всего команды `test` и `[[ ]]` поддерживают три логические операции. Это И (AND), ИЛИ (OR) и НЕ

(NOT). Для представления этих операций `test` и `[[ ]]` используют разные операторы, как показано в табл. 27.4.

Таблица 27.4. Логические операторы

Операция	test	[[ ]] и (( ))
И	-a	&&
ИЛИ	-o	||
НЕ	!	!

Ниже приводится пример использования операции И (AND). Следующий сценарий определяет вхождение целочисленного значения в определенный диапазон:

```
#!/bin/bash

# test-integer3: проверка вхождения целочисленного значения
# в определенный диапазон.

MIN_VAL=1
MAX_VAL=100

INT=50

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
      if [[ "$INT" -ge "$MIN_VAL" && "$INT" -le "$MAX_VAL" ]]; then
            echo "$INT is within $MIN_VAL to $MAX_VAL."
      else
            echo "$INT is out of range."
      fi
else
      echo "$INT is not an integer." >&2
      exit 1
fi
```

Этот сценарий определяет, попадает ли целочисленное значение `INT` в диапазон между `MIN_VAL` и `MAX_VAL`. Эта операция выполняется единственной командой `[[ ]]`, включающей два выражения, разделенных оператором `&&`. Ту же проверку можно выполнить с помощью `test`:

```
if [ "$INT" -ge "$MIN_VAL" -a "$INT" -le "$MAX_VAL" ]; then
     echo "$INT is within $MIN_VAL to $MAX_VAL."
else
    echo "$INT is out of range."
fi
```

Оператор отрицания `!` обращает результат выражения. Он возвращает истинное значение, если выражение ложно, и ложное значение, если выражение истинно. В следующем сценарии мы изменили логику вычислений, чтобы определить, находится ли значение `INT` за пределами указанного диапазона:

```
#!/bin/bash

# test-integer4: проверка выхода целочисленного значения
# за границы определенного диапазона.

MIN_VAL=1
MAX_VAL=100

INT=50

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
      if [[ ! ("$INT" -ge "$MIN_VAL" && "$INT" -le "$MAX_VAL") ]]; then
            echo "$INT is outside $MIN_VAL to $MAX_VAL."
      else
            echo "$INT is in range."
      fi
else
      echo "$INT is not an integer." >&2
      exit 1
fi
```

Здесь выражение заключено в круглые скобки для группировки. Если этого не сделать, оператор отрицания будет применяться к результату первого выражения, а не к объединению двух выражений. Ту же проверку можно реализовать с помощью `test`:

```
if [ ! \( "$INT" -ge "$MIN_VAL" -a "$INT" -le "$MAX_VAL" \) ]; then
      echo "$INT is outside $MIN_VAL to $MAX_VAL."
else
      echo "$INT is in range."
fi
```

Поскольку все выражения и операторы в команде `test` интерпретируются командной оболочкой как аргументы (в отличие от `[[ ]]` и `(( ))`), символы, имеющие специальное значение для `bash`, такие как `<`, `>`, `(` и `)`, необходимо заключать в кавычки или экранировать.

Учитывая, что команды `test` и `[[ ]]` до определенной степени равноценны, возникает вопрос: какой из них отдать предпочтение? Команда `test` является традиционной (и частью стандарта POSIX), тогда как команда `[[ ]]` характерна для `bash`. Уметь пользоваться командой `test` крайне важно, потому что она применяется

очень широко, но команда `[[ ]]` проще и удобнее, поэтому она часто используется в современных сценариях.

> **ПЕРЕНОСИМОСТЬ — БЕСПОЧВЕННЫЕ СТРАХИ ОТ НЕПОНИМАНИЯ**
>
> Если вам доведется побеседовать с «истинными» пользователями Unix, вы быстро обнаружите, что многие из них Linux терпеть не могут. Они оценивают его как нечто нечистое и греховное. Один из принципов таких ревнителей Unix — все должно быть «переносимым». То есть любой сценарий, написанный вами, должен работать без изменений в любой Unix-подобной системе.
>
> Пользователи Unix имеют веские основания рассчитывать на это. Наблюдая последствия для мира Unix, вызванные внедрением проприетарных расширений команд и командных оболочек до появления POSIX, они естественно опасаются влияния Linux на их любимую ОС.
>
> Но *переносимость* имеет серьезный недостаток. Она тормозит прогресс и требует приведения всего и вся к «наименьшему общему знаменателю». Для сценариев на языке командной оболочки это означает, что они должны быть совместимы с `sh`, оригинальной командной оболочкой Bourne.
>
> Этот недостаток служит отговоркой, которой пользуются производители проприетарных расширений для их оправдания, только они называют их «новшествами». Но в действительности они замыкают пользователей на себя.
>
> Инструменты GNU, такие как `bash`, не имеют подобных ограничений. Они способствуют переносимости благодаря поддержке стандартов и всеобщей доступности. `bash` и другие инструменты GNU можно установить практически в любую систему, даже в Windows, совершенно бесплатно. Поэтому не бойтесь использовать все возможности, имеющиеся в командной оболочке `bash`. Она *действительно* переносима.

Операторы управления: еще один способ ветвления

`bash` поддерживает два оператора управления, которые используются для ветвления. Операторы `&&` (И) и `||` (ИЛИ) действуют подобно логическим операторам в составной команде `[[ ]]`. Они имеют следующий синтаксис:

```
команда1 && команда2
```

и

```
команда1 || команда2
```

Важно понимать, как они действуют. В последовательности с оператором `&&` первая команда выполняется всегда, а вторая — *только если первая завершилась успехом*. В последовательности с оператором `||` первая команда выполняется всегда, а вторая — *только если первая завершилась неудачей*.

В практическом смысле это означает, что можно выполнить следующую последовательность команд:

```
[me@linuxbox ~]$ mkdir temp && cd temp
```

Она создаст каталог с именем *temp* и, если эта операция завершится успехом, каталог *temp* будет назначен текущим рабочим каталогом. Попытка выполнить вторую команду будет произведена, только если команда `mkdir` завершится успехом. Аналогично, следующая команда

```
[me@linuxbox ~]$ [ -d temp ] || mkdir temp
```

проверит существование каталога *temp*, и только если проверка не увенчается успехом, будет выполнена команда его создания. Такие конструкции очень удобно использовать для обработки ошибок в сценариях, о чем подробнее рассказывается в следующих главах. Например, в сценарии можно предусмотреть такую последовательность:

```
[ -d temp ] || exit 1
```

Если сценарий требует наличия каталога *temp*, а он не существует, тогда сценарий завершится с кодом 1.

Заключение

Мы начали эту главу с вопроса, оставшегося без ответа в предыдущей главе: как сценарию `sys_info_page` определить, имеет ли текущий пользователь права на чтение всех домашних каталогов? После знакомства с инструкцией `if` эту проблему можно решить, добавив следующий код в функцию `report_home_space`:

```
report_home_space () {
      if [[ $(id -u) -eq 0 ]]; then
            cat <<- _EOF_
                  <h2>Home Space Utilization (All Users)</h2>
                  <pre>$(du -sh /home/*)</pre>
                  _EOF_
      else
            cat <<- _EOF_
            <h2>Home Space Utilization ($USER)</h2>
```

```
            <pre>$(du -sh $HOME)</pre>
            _EOF_
      fi
      return
}
```

Здесь проверяется вывод команды `id`. Если вызвать команду `id` с параметром `-u`, она выведет числовой идентификатор действующего пользователя. Суперпользователю всегда присваивается числовой идентификатор 0. Зная это, мы сконструировали два разных вложенных документа: один пользуется преимуществом привилегий суперпользователя, а другой ограничивается домашним каталогом текущего пользователя.

Теперь мы немного отдохнем от программы `sys_info_page`, но не волнуйтесь. Мы еще вернемся к ней. А пока затронем те темы, знание которых потребуется, когда мы возобновим разработку.

28 Чтение ввода с клавиатуры

В сценариях, написанных нами до сих пор, отсутствует одно свойство, характерное для многих компьютерных программ, — *интерактивность*, то есть возможность взаимодействия с пользователем. Несмотря на то что многие программы не нуждаются в интерактивности, некоторые только выиграли бы, если бы имели возможность принимать ввод непосредственно от пользователя. Возьмем для примера сценарий из предыдущей главы:

```
#!/bin/bash

# test-integer2: проверка целочисленного значения.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then
      if [ "$INT" -eq 0 ]; then
            echo "INT is zero."
      else
            if ["$INT" -lt 0 ]; then
                  echo "INT is negative."
            else
                  echo "INT is positive."
            fi
            if [ $((INT % 2)) -eq 0 ]; then
                  echo "INT is even."
            else
                  echo "INT is odd."
            fi
      fi
else
      echo "INT is not an integer." >&2
      exit 1
fi
```

Каждый раз, когда потребуется изменить значение `INT`, вы должны будете изменить сценарий. Пользоваться сценарием было бы удобнее, если бы он предлагал пользователю ввести значение. В этой главе мы посмотрим, как придать интерактивность нашим программам.

read — чтение значений со стандартного ввода

Встроенная команда `read` используется для чтения единственной строки со стандартного ввода. Эту команду можно использовать для чтения ввода с клавиатуры или, в случае перенаправления, строки данных из файла. Команда имеет следующий синтаксис:

```
read [-параметры] [переменная...]
```

где ***параметры*** — это один или несколько параметров из перечисленных в табл. 28.1 далее, а ***переменная*** — имя одной или нескольких переменных для сохранения введенного значения. Если имя переменной не указано, строка с данными сохраняется в переменной `REPLY`.

В простейшем случае `read` сохраняет значения полей, прочитанные со стандартного ввода, в указанные переменные. Ниже показано, как можно изменить наш сценарий проверки целочисленных значений, задействовав в нем команду `read`:

```
#!/bin/bash

# read-integer: проверка целочисленного значения.

echo -n "Please enter an integer -> "
read int

if [[ "$int" =~ ^-?[0-9]+$ ]]; then
      if [ "$int" -eq 0 ]; then
            echo "$int is zero."
      else
            if [ "$int" -lt 0 ]; then
                  echo "$int is negative."
            else
                  echo "$int is positive."
            fi
            if [ $((int % 2)) -eq 0 ]; then
                  echo "$int is even."
            else
                  echo "$int is odd."
            fi
```

```
      fi
else
      echo "Input value is not an integer." >&2
      exit 1
fi
```

Сначала мы использовали команду `echo` с параметром `-n` (подавляющим вывод символа перевода строки в конце) для вывода приглашения к вводу, а затем команду `read` для ввода значения в переменную `int`. Запуск этого сценария приводит к следующим результатам:

```
[me@linuxbox ~]$ read-integer
Please enter an integer -> 5
5 is positive.
5 is odd.
```

Команда `read` может сохранять ввод в множестве переменных, это показано в следующем сценарии:

```
#!/bin/bash

# read-multiple: чтение нескольких значений с клавиатуры

echo -n "Enter one or more values > "
read var1 var2 var3 var4 var5

echo "var1 = '$var1'"
echo "var2 = '$var2'"
echo "var3 = '$var3'"
echo "var4 = '$var4'"
echo "var5 = '$var5'"
```

Этот сценарий вводит, присваивает переменным и выводит до пяти значений. Обратите внимание, как действует команда `read`, когда получает разное число значений:

```
[me@linuxbox ~]$ read-multiple
Enter one or more values > a b c d e
var1 = 'a'
var2 = 'b'
var3 = 'c'
var4 = 'd'
var5 = 'e'
[me@linuxbox ~]$ read-multiple
Enter one or more values > a
var1 = 'a'
```

```
var2 = ''
var3 = ''
var4 = ''
var5 = ''
[me@linuxbox ~]$ read-multiple
Enter one or more values > a b c d e f g
var1 = 'a'
var2 = 'b'
var3 = 'c'
var4 = 'd'
var5 = 'e f g'
```

Если `read` получит меньше значений, чем ожидается, переменные, для которых не хватило значений, останутся пустыми, а при избыточном количестве значений на входе последняя переменная получит весь остаток введенной строки.

Если не передать переменные команде `read`, весь ввод будет сохранен в переменной командной оболочки `REPLY`:

```
#!/bin/bash

# read-single: чтение множества значений в переменную по умолчанию

echo -n "Enter one or more values > "
read

echo "REPLY = '$REPLY'"
```

Запуск этого сценария приводит к следующим результатам:

```
[me@linuxbox ~]$ read-single
Enter one or more values > a b c d
REPLY = 'a b c d'
```

Параметры

`read` поддерживает параметры, перечисленные в табл. 28.1.

Таблица 28.1. Параметры команды read

Параметр	Описание
`-a массив`	Сохранить ввод в указанный массив, начиная с элемента с индексом 0. Подробнее о массивах рассказывается в главе 35
`-d разделитель`	Использовать в качестве признака конца ввода первый символ из строки разделитель, а не символ перевода строки

Параметр	Описание
`-e`	Использовать Readline для обработки ввода. Это позволяет редактировать ввод, как в командной строке
`-n число`	Прочитать указанное число символов, а не всю строку
`-p приглашение`	Показывать указанное приглашение к вводу
`-r`	Режим без промежуточной обработки. Не интерпретировать обратные слеши как экранирующие символы
`-s`	Безмолвный режим. Не производить эхо-вывод символов на экран в процессе ввода. Этот режим может пригодиться для организации ввода паролей и другой конфиденциальной информации
`-t секунды`	Предельное время ожидания. Завершить ввод по истечении указанного числа секунд. По истечении указанного интервала `read` вернет ненулевое значение
`-u дескриптор`	Произвести ввод из файла с указанным дескриптором вместо стандартного ввода

Множество поддерживаемых параметров открывает доступ к довольно интересным способам использования `read`. Например, параметр `-p` позволяет определить строку приглашения к вводу:

```
#!/bin/bash

# read-single: чтение множества значений в переменную по умолчанию

read -p "Enter one or more values > "

echo "REPLY = '$REPLY'"
```

Параметры `-t` и `-s` позволяют писать сценарии, реализующие ввод «секретных» данных и прерывающие ввод по истечении заданного времени:

```
#!/bin/bash

# read-secret: ввод секретного пароля

if read -t 10 -sp "Enter secret passphrase > " secret_pass; then
      echo -e "\nSecret passphrase = '$secret_pass'"
else
      echo -e "\nInput timed out" >&2
      exit 1
fi
```

Сценарий предлагает пользователю ввести секретный пароль и ждет 10 секунд. Если в течение этого времени ввод не был завершен, сценарий завершается с кодом ошибки. Поскольку в команду включен параметр `-s`, символы пароля не выводятся на экран в процессе ввода.

С помощью параметров `-e` и `-i` можно предложить пользователю значение по умолчанию для ввода.

```
#!/bin/bash

# read-default: предложение значения по умолчанию, для ввода
#               которого достаточно нажать клавишу Enter.

read -e -p "What is your user name? " -i $USER
echo "You answered: '$REPLY'"
```

Этот сценарий просит ввести имя учетной записи и использует переменную окружения `USER`, чтобы предложить значение по умолчанию. После запуска сценарий выведет строку со значением по умолчанию, и если пользователь просто нажмет `ENTER`, команда `read` присвоит это значение переменной `REPLY`.

```
[me@linuxbox ~]$ read-default
What is your user name? me
You answered: 'me'
```

Выделение полей в строке ввода с помощью IFS

Обычно командная оболочка разбивает введенную строку на слова перед передачей команде `read`. Как мы уже знаем, в результате этого слова в строке, разделенные одним или несколькими пробелами, превращаются в отдельные значения и присваиваются командой `read` разным переменным. Такое поведение командной оболочки регулируется переменной с именем `IFS` (от Internal Field Separator — внутренний разделитель полей). По умолчанию переменная `IFS` хранит символы пробела, табуляции и перевода строки, каждый из которых может служить разделителем полей.

Изменяя значение переменной `IFS`, можно управлять делением ввода на поля перед передачей команде `read`. Например, файл */etc/passwd* хранит строки данных, в которых поля отделяются друг от друга двоеточием. Присвоив переменной `IFS` значение, состоящее из единственного двоеточия, можно с помощью `read` прочитать содержимое */etc/passwd* и благополучно разделить строки на поля для присваивания разным переменным. Ниже приводится сценарий, который именно так и действует:

```
#!/bin/bash

# read-ifs: чтение полей из файла

FILE=/etc/passwd

read -p "Enter a username > " user_name

file_info=$(grep "^$user_name:" $FILE)

if [ -n "$file_info" ]; then
      IFS=":" read user pw uid gid name home shell <<< "$file_info"
      echo "User =      '$user'"
      echo "UID =       '$uid'"
      echo "GID =       '$gid'"
      echo "Full Name = '$name'"
      echo "Home Dir. = '$home'"
      echo "Shell =     '$shell'"
else
      echo "No such user '$user_name'" >&2
      exit 1
fi
```

Этот сценарий предлагает пользователю ввести имя учетной записи в системе и затем выводит разные поля, найденные в соответствующей записи в файле */etc/passwd*. В сценарии есть две интересные строки. Первая:

```
file_info=$(grep "^$user_name:" $FILE)
```

присваивает результат команды `grep` переменной `file_info`. Регулярное выражение гарантирует извлечение из файла */etc/passwd* единственной строки, соответствующей введенному имени пользователя.

Вторая интересная строка:

```
IFS=":" read user pw uid gid name home shell <<< "$file_info"
```

состоит из трех частей: присваивания значения переменной, команды `read` со списком имен переменных в виде аргументов и незнакомого нам, нового оператора перенаправления. Рассмотрим сначала присваивание значения переменной.

Командная оболочка позволяет выполнить одну или несколько операций присваивания значений переменным непосредственно перед командой, в той же строке. Они изменяют окружение, в котором выполняется команда. Действие этих

операций присваивания носит временный характер, окружение изменяется только на время выполнения команды. В данном случае в переменной `IFS` сохраняется двоеточие. То же самое можно выразить иначе:

```
OLD_IFS="$IFS"
IFS=":"
read user pw uid gid name home shell <<< "$file_info"
IFS="$OLD_IFS"
```

Здесь мы сохранили прежнее значение `IFS`, присвоили новое значение, выполнили команду `read` и восстановили прежнее значение `IFS`. Очевидно, что размещение операции присваивания перед командой позволяет получить более компактный код, действующий точно так же.

Оператор <<< отмечает *встроенную строку*. Встроенная строка (here string) подобна встроенному документу, только короче, она простирается лишь до конца текущей строки кода. В данном примере строка с данными из файла */etc/passwd* подается на стандартный ввод команды `read`. У кого-то может возникнуть вопрос, почему был выбран такой, несколько необычный, способ вместо

```
echo "$file_info" | IFS=":" read user pw uid gid name home shell
```

Скажем так: на то есть свои причины...

Проверка ввода

Использование новой для нас возможности приема ввода с клавиатуры влечет за собой дополнительную проблему: необходимость проверки введенных данных. Очень часто хорошо написанная программа отличается от плохо написанной готовностью к неожиданностям. Зачастую неожиданности возникают в форме ввода ошибочных данных. Мы уже сделали кое-что, чтобы противостоять неожиданностям в программах проверки целочисленных значений из предыдущей главы, где предусмотрено отсеивание пустых значений и значений с нецифровыми символами. Такого рода программные проверки должны выполняться для любых вводимых данных, чтобы обезопасить программу от недопустимых значений. Это особенно актуально для программ, используемых множеством пользователей. Отказ от защитных мер ради экономии простителен, только если программа пишется для однократного использования автором с целью решения некоей специальной задачи. Но даже в этом случае, если программа выполняет потенциально опасные операции, такие как удаление файлов, на всякий случай включите в нее проверку данных.

READ НЕЛЬЗЯ ИСПОЛЬЗОВАТЬ В КОНВЕЙЕРЕ

Даже притом, что команда `read` способна принимать данные со стандартного ввода, она не позволяет использовать ее следующим образом:

```
echo "foo" | read
```

Можно было бы ожидать, что этот прием сработает, но это не так. Внешне все будет выглядеть так, как будто команда успешно отработала, но при этом переменная `REPLY` всегда будет оставаться пустой. Почему?

Объясняется это особенностью обработки конвейеров командной оболочкой. В `bash` (и в других командных оболочках, таких как `sh`) конвейеры создают *подоболочки* (subshells). Они являются копиями родительской оболочки и ее окружения и используются для выполнения команд в конвейерах. В предыдущем примере команда `read` выполняется в подоболочке.

Для подоболочек в Unix-подобных системах создаются копии родительского окружения, которые они и используют в работе. Когда конвейер завершается, копия окружения уничтожается. Это означает, что *подоболочка никогда не сможет изменить окружение родительского процесса*. Как мы знаем, `read` присваивает значения переменным, которые становятся частью окружения. В примере выше `read` присвоит значение `foo` переменной `REPLY` в окружении подоболочки, но когда конвейер завершится, подоболочка и ее окружение будут уничтожены, а результат присваивания будет утрачен.

Использование встроенных строк — один из способов обойти эту проблему. Еще один способ мы увидим в главе 36.

Далее приводится пример программы, проверяющий входные данные разного вида:

```
#!/bin/bash

# read-validate: проверка ввода

invalid_input () {
      echo "Invalid input '$REPLY'" >&2
      exit 1
}

read -p "Enter a single item > "

# пустой ввод (недопустимо)
[[ -z "$REPLY" ]] && invalid_input
```

```
# ввод множества элементов (недопустимо)
(( "$(echo "$REPLY" | wc -w)" > 1 )) && invalid_input

# введено допустимое имя файла?
if [[ "$REPLY" =~ ^[-[:alnum:]\._]+$ ]]; then
      echo "'$REPLY' is a valid filename."
      if [[ -e "$REPLY" ]]; then
            echo "And file '$REPLY' exists."
      else
            echo "However, file '$REPLY' does not exist."
      fi

      # введено вещественное число?
      if [[ "$REPLY" =~ ^-?[[:digit:]]*\.[[:digit:]]+$ ]]; then
            echo "'$REPLY' is a floating point number."
      else
            echo "'$REPLY' is not a floating point number."
      fi

      # введено целое число?
      if [[ "$REPLY" =~ ^-?[[:digit:]]+$ ]]; then
            echo "'$REPLY' is an integer."
      else
            echo "'$REPLY' is not an integer."
      fi
else
      echo "The string '$REPLY' is not a valid filename."
fi
```

Этот сценарий предлагает пользователю ввести элемент данных и затем последовательно анализирует его содержимое. Как видите, в сценарии использовано множество идей, с которыми мы уже познакомились, включая функции, `[[ ]]`, `(( ))`, операторы управления `&&` и `if`, а также разумную дозу регулярных выражений.

Меню

Часто для организации интерактивной работы используются *меню*. Программы, управляемые системой меню, выводят список возможных вариантов и предлагают пользователю выбрать один из них. Например, представьте программу, которая выводит следующее:

```
Выберите команду:

1. Вывести информацию о системе
2. Вывести информацию о дисковом пространстве
3. Вывести информацию об объеме домашнего каталога
```

```
0. Выйти

Введите номер выбранной команды [0-3] >
```

Используя все, что мы узнали в ходе создания программы `sys_info_page`, можно сконструировать программу, реализующую решение задач, перечисленных в меню, приведенном выше:

```
#!/bin/bash

# read-menu: программа вывода системной информации,
#            управляемая с помощью меню

clear
echo "
Please Select:

1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"

read -p "Enter selection [0-3] > "

if [[ "$REPLY" =~ ^[0-3]$ ]]; then
      if [[ "$REPLY" == 0 ]]; then
            echo "Program terminated."
            exit
      fi
      if [[ "$REPLY" == 1 ]]; then
            echo "Hostname: $HOSTNAME"
            uptime
            exit
      fi
      if [[ "$REPLY" == 2 ]]; then
            df -h
            exit
      fi
      if [[ "$REPLY" == 3 ]]; then
            if [[ "$(id -u)" -eq 0 ]]; then
                  echo "Home Space Utilization (All Users)"
                  du -sh /home/*
            else
                  echo "Home Space Utilization ($USER)"
                  du -sh "$HOME"
            fi
```

```
            exit
      fi
else
      echo "Invalid entry." >&2
      exit 1
fi
```

Этот сценарий делится на две логические части. Первая часть выводит меню и вводит выбор пользователя. Вторая часть идентифицирует выбор и выполняет соответствующие действия. Обратите внимание, как используется команда `exit` в этом сценарии. Она препятствует выполнению ненужного кода после завершения затребованного действия. Наличие нескольких точек выхода из программы вообще считается дурным тоном (логику работы такой программы труднее понять), но в данном сценарии нас это устраивает.

Заключение

В этой главе мы сделали первый шаг к интерактивности, позволив пользователю вводить данные в наши программы с клавиатуры. Используя описанные приемы, можно написать множество полезных программ, например, программы, выполняющие специализированные вычисления или упрощающие доступ к таинственным инструментам командной строки. В следующей главе мы усовершенствуем идею программ, управляемых при помощи меню, чтобы добиться большего.

Дополнительные сведения

Постарайтесь внимательно изучить программы из этой главы и достичь полного понимания их логической структуры, потому что программы, которые последуют далее, будут еще сложнее. В качестве упражнения перепишите программы этой главы, используя команду `test` вместо составной команды `[[ ]]`. Подсказка: используйте `grep` для сопоставления с регулярными выражениями, а затем проверяйте код завершения. Это станет для вас хорошей практикой.

29 Управление потоком выполнения: циклы while и until

В предыдущей главе мы написали программу, управляемую с помощью меню, для получения разного рода системной информации. Программа работает, но неудобна в использовании. Она выполняет только один выбранный вариант и завершается. Хуже того, в случае ошибочного выбора программа завершается с выводом сообщения об ошибке, не давая возможности повторить попытку. Пользоваться программой было бы намного удобнее, если бы она снова и снова выводила меню и предлагала сделать выбор, пока пользователь не выберет пункт, соответствующий выходу из программы.

В этой главе мы познакомимся с приемами организации *циклов*, с помощью которых можно реализовать многократное выполнение участков программ. Командная оболочка поддерживает три составные команды для организации циклов. Здесь мы познакомимся с двумя из них, а с третьей — в главе 33.

Циклы

Повседневная жизнь наполнена повторяющимися действиями. Каждодневная поездка на работу, прогулка с собакой и нарезание моркови — все эти действия состоят из повторяющейся последовательности действий. Рассмотрим в качестве примера резку моркови. Этот вид деятельности можно выразить на псевдокоде примерно так:

1. Взять разделочную доску.
2. Взять нож.
3. Положить морковь на доску.
4. Поднять нож.
5. Сдвинуть морковь.

6. Отрезать кусок.
7. Если вся морковь порезана, завершить операцию, иначе перейти к шагу 4.

Шаги с 4-го по 7-й образуют *цикл*. Действия внутри цикла повторяются, пока не будет выполнено условие «вся морковь порезана».

while

В `bash` имеются средства, позволяющие выражать похожие идеи. Представьте, что нам нужно вывести пять чисел по порядку, от 1 до 5. В сценарии на языке `bash` это можно реализовать, как показано ниже:

```
#!/bin/bash

# while-count: вывод последовательности чисел

count=1

while [[ $count -le 5 ]]; do
      echo $count
      count=$((count + 1))
done
echo "Finished."
```

Если запустить этот сценарий, он выведет:

```
[me@linuxbox ~]$ while-count
1
2
3
4
5
Finished.
```

Команда `while` имеет следующий синтаксис:

```
while команды; do команды; done
```

Подобно `if`, команда `while` проверяет код завершения списка команд. Пока код завершения равен 0, она выполняет команды внутри цикла. В сценарии, приведенном выше, создается переменная `count`, и ей присваивается начальное значение 1. Команда `while` проверяет код завершения составной команды `[[ ]]`. Пока `[[ ]]` возвращает код 0, команды внутри цикла продолжают выполняться. В конце каждого цикла повторно выполняется команда `[[ ]]`. После пяти итераций цикла значение переменной `count` увеличится до 6, команда `[[ ]]` вернет код завершения,

отличный от 0, и цикл завершится, а программа продолжит выполнение с инструкции, следующей непосредственно за циклом.

Цикл `while` можно использовать для усовершенствования программы `read-menu` из предыдущей главы:

```
#!/bin/bash

# while-menu: программа вывода системной информации,
#             управляемая с помощью меню

DELAY=3 # Время отображения результатов на экране (в секундах)

while [[ "$REPLY" != 0 ]]; do
      clear
      cat <<- _EOF_
            Please Select:

            1. Display System Information
            2. Display Disk Space
            3. Display Home Space Utilization
            0. Quit

      _EOF_
      read -p "Enter selection [0-3] > "

      if [[ "$REPLY" =~ ^[0-3]$ ]]; then
            if [[ "$REPLY" == 1 ]]; then
                  echo "Hostname: $HOSTNAME"
                  uptime
                  sleep "$DELAY"
            fi
            if [[ "$REPLY" == 2 ]]; then
                  df -h
                  sleep "$DELAY"
            fi
            if [[ "$REPLY" == 3 ]]; then
                  if [[ "$(id -u)" -eq 0 ]]; then
                        echo "Home Space Utilization (All Users)"
                        du -sh /home/*
                  else
                        echo "Home Space Utilization ($USER)"
                        du -sh "$HOME"
                  fi
                  sleep "$DELAY"
            fi
      else
```

```
            echo "Invalid entry."
            sleep "$DELAY"
      fi
done
echo "Program terminated."
```

Заключив меню в цикл `while`, мы смогли заставить программу повторять вывод меню после каждой операции выбора. Цикл продолжает выполняться и выводить меню, пока переменная `REPLY` не получит значение 0, предоставляя пользователю возможность сделать другой выбор. После выполнения выбранной операции выполняется команда `sleep`, она приостанавливает программу на несколько секунд и дает возможность увидеть результаты до того, как экран будет очищен и на нем вновь появится меню. Когда переменная `REPLY` получит значение 0, соответствующее варианту «Quit» (выйти), цикл завершится и выполнение продолжится со строки, следующей за `done`.

Прерывание цикла

В `bash` имеются две встроенные команды для управления потоком выполнения внутри циклов. Команда `break` немедленно завершает цикл, после чего выполнение программы продолжается с первой инструкции, следующей за циклом. Команда пропускает оставшуюся часть цикла, и программа переходит к началу следующей итерации цикла. Ниже приводится версия программы `while-menu`, использующая обе команды — `break` и `continue`:

```
#!/bin/bash

# while-menu2: программа вывода системной информации,
#              управляемая с помощью меню

DELAY=3 # Время отображения результатов на экране (в секундах)

while true; do
      clear
      cat <<- _EOF_
            Please Select:

            1. Display System Information
            2. Display Disk Space
            3. Display Home Space Utilization
            0. Quit
      _EOF_
      read -p "Enter selection [0-3] > "
```

```
    if [[ "$REPLY" =~ ^[0-3]$ ]]; then
        if [[ "$REPLY" == 1 ]]; then
            echo "Hostname: $HOSTNAME"
            uptime
            sleep "$DELAY"
            continue
        fi
        if [[ "$REPLY" == 2 ]]; then
            df -h
            sleep "$DELAY"
            continue
        fi
        if [[ "$REPLY" == 3 ]]; then
            if [[ "$(id -u)" -eq 0 ]]; then
                echo "Home Space Utilization (All Users)"
                du -sh /home/*
            else
                echo "Home Space Utilization ($USER)"
                du -sh "$HOME"
            fi
            sleep "$DELAY"
            continue
        fi
        if [[ "$REPLY" == 0 ]]; then
            break
        fi
    else
        echo "Invalid entry."
        sleep $DELAY
    fi
done
echo "Program terminated."
```

В этой версии сценария используется *бесконечный цикл* (цикл, который никогда не завершится сам по себе), в котором команда `while` проверяет код завершения команды `true`. Так как `true` всегда возвращает код 0, цикл никогда не завершится. Этот прием на удивление широко используется в сценариях. Поскольку цикл никогда не завершится сам по себе, программист должен предусмотреть его принудительное прерывание в нужный момент времени. В этом сценарии выход из цикла осуществляется с помощью команды `break`, когда пользователь выберет пункт 0. В конец других операций добавлена команда `continue`, чтобы увеличить эффективность работы сценария. Встретив команду `continue`, сценарий перепрыгнет через остальной код в цикле, который не требуется выполнять для данного выбора. Например, если пользователь выбрал пункт 1, нет никаких причин проверять выбор остальных вариантов.

until

Команда `until` очень похожа на `while`, но завершает цикл не когда обнаружит ненулевой код завершения, а наоборот. Цикл `until` продолжается, пока не получит код завершения 0. В сценарии `while-count` цикл продолжает выполняться, пока значение переменной `count` меньше или равно 5. Тот же результат можно получить, переписав сценарий с командой `until`:

```
#!/bin/bash

# until-count: вывод последовательности чисел

count=1

until [[ "$count" -gt 5 ]]; do
      echo "$count"
      count=$((count + 1))
done
echo "Finished."
```

С условным выражением `$count -gt 5` команда `until` завершит цикл в нужный момент времени. Выбор между циклами `while` и `until` обычно зависит от того, в каком случае условное выражение будет более читабельным.

Чтение файлов в циклах

Команды `while` и `until` могут принимать данные со стандартного ввода. Это дает возможность обрабатывать файлы с их помощью. В следующем примере мы выведем содержимое файла *distros.txt*, созданного в одной из предыдущих глав:

```
#!/bin/bash

# while-read: чтение строк из файла

while read distro version release; do
      printf "Distro: %s\tVersion: %s\tReleased: %s\n" \
      "$distro" \
      "$version" \
      "$release"
done < distros.txt
```

Чтобы перенаправить файл в цикл, мы поместили оператор перенаправления после инструкции `done`. Цикл будет вводить поля из указанного файла с помощью `read`. После ввода каждой строки команда `read` будет завершаться с кодом 0, пока

не достигнет конца файла. В этот момент она вернет ненулевой код завершения, и цикл завершится. Цикл можно также использовать в конвейерах:

```
#!/bin/bash

# while-read2: чтение строк из файла

sort -k 1,1 -k 2n distros.txt | while read distro version release; do
      printf "Distro: %s\tVersion: %s\tReleased: %s\n" \
            "$distro" \
            "$version" \
            "$release"
done
```

Здесь вывод команды `sort` передается на стандартный ввод цикла, который выводит поток текста на экран. Но не забывайте, что конвейер выполняет цикл в подоболочке, поэтому после его завершения любые переменные, созданные в цикле, будут потеряны.

Заключение

После знакомства с циклами и ранее представленными командами ветвления, функциями и последовательностями мы получили представление об основных способах управления потоком выполнения в программах. В арсенале `bash` имеется еще множество хитрых трюков, но все они основаны на этих простых идеях.

30 Поиск и устранение ошибок

Поскольку наши сценарии становятся все сложнее и сложнее, настало время посмотреть, что случается, когда что-то идет не так и сценарии перестают делать то, что нам нужно. В этой главе мы познакомимся с некоторыми распространенными ошибками, встречающимися в сценариях, и приемами поиска и устранения неисправностей.

Синтаксические ошибки

Один из самых распространенных видов ошибок — синтаксические ошибки. Синтаксические ошибки возникают при неправильном вводе некоторого элемента с нарушением синтаксиса командной оболочки. Чаще всего эти ошибки вызывают отказ командной оболочки от выполнения сценария.

Для демонстрации распространенных видов ошибок в дальнейших обсуждениях мы будем использовать следующий сценарий:

```
#!/bin/bash

# trouble: сценарий для демонстрации распространенных видов ошибок

number=1

if [ $number = 1 ]; then
      echo "Number is equal to 1."
else
      echo "Number is not equal to 1."
fi
```

В текущем своем виде сценарий выполняется без ошибок:

```
[me@linuxbox ~]$ trouble
Number is equal to 1.
```

Отсутствующие кавычки

Давайте изменим сценарий, удалив кавычку в конце аргумента первой команды `echo`:

```
#!/bin/bash

# trouble: сценарий для демонстрации распространенных видов ошибок

number=1

if [ $number = 1 ]; then
      echo "Number is equal to 1.
else
      echo "Number is not equal to 1."
fi
```

Посмотрите, что из этого получилось:

```
[me@linuxbox ~]$ trouble
./trouble: строка 10: неожиданный EOF при поиске соответствующего `"'
./trouble: строка 13: ошибка синтаксиса: неожиданный конец файла
```

Командная оболочка сгенерировала два сообщения об ошибках. Обратите внимание, что номера строк в сообщениях не соответствуют номеру строки, где отсутствует кавычка. Понять причину можно, мысленно последовав за программой после отсутствующей кавычки. `bash` продолжит поиск закрывающей кавычки и найдет ее сразу за второй командой `echo`. После этого командная оболочка `bash` очень удивится, обнаружив нарушение синтаксиса команды `if`, потому что инструкция `fi` теперь окажется внутри строки в кавычках (незакрытой).

Найти такие ошибки в длинных сценариях порой очень сложно. Хорошую помощь в этом случае может оказать текстовый редактор с подсветкой синтаксиса. Если в системе установлена полная версия редактора `vim`, подсветка синтаксиса в нем включается командой:

```
:syntax on
```

Отсутствующие или неожиданные лексемы

Другая частая ошибка — отсутствие закрывающего элемента в составной команде, такой как `if` или `while`. Взгляните, что получится, если убрать точку с запятой после проверки условия в команде `if`:

```
#!/bin/bash

# trouble: сценарий для демонстрации распространенных видов ошибок

number=1

if [ $number = 1 ] then
      echo "Number is equal to 1."
else
      echo "Number is not equal to 1."
fi
```

При попытке выполнить сценарий мы получим:

```
[me@linuxbox ~]$ trouble
./trouble: строка 9: ошибка синтаксиса около неожиданной лексемы `else'
./trouble: строка 9: `else'
```

И снова сообщение об ошибке указывает на место, расположенное гораздо дальше фактического места ошибки. Здесь складывается очень интересная ситуация. Как вы помните, `if` принимает список команд и проверяет код завершения последней команды в списке. В нашей программе мы задумали список с единственной командой `[`, которая является синонимом команды `test`. Команда `[` принимает все, что следует за ней, как список аргументов — в данном случае четыре аргумента: `$number`, `=`, `1` и `]`. В отсутствие точки с запятой в список аргументов будет добавлено слово `then`, что синтаксически допустимо. Следующая команда `echo` также допустима. Она интерпретируется как еще одна команда в списке команд, которую `if` должна выполнить и проверить код завершения. Далее следует неуместное здесь слово `else`, потому что командная оболочка распознает его как *зарезервированное слово* (слово, имеющее специальное значение для командной оболочки), а не как имя команды. Это объясняет смысл сообщения об ошибке.

Непредвиденная подстановка

Существуют ошибки, которые возникают лишь время от времени. Иногда сценарий работает без ошибок, а иногда терпит неудачу из-за работы механизма

подстановки. Для демонстрации этой проблемы вернем точку с запятой на место и изменим значение переменной `number`, присвоив ей пустое значение:

```
#!/bin/bash

# trouble: сценарий для демонстрации распространенных видов ошибок

number=

if [ $number = 1 ]; then
      echo "Number is equal to 1."
else
      echo "Number is not equal to 1."
fi
```

При попытке выполнить сценарий после внесения изменений мы получим:

```
[me@linuxbox ~]$ trouble
./trouble: строка 7: [: =: ожидается использование унарного оператора
Number is not equal to 1.
```

Мы получили довольно загадочное сообщение, за которым следует вывод второй команды `echo`. Проблема заключается в подстановке переменной `number` в команду `test`. После обработки команды

```
[ $number = 1 ]
```

механизмом подстановки, который заменит `number` пустым значением:

```
[ = 1 ]
```

получится недопустимый результат, и командная оболочка сгенерирует сообщение об ошибке. Оператор `=` является бинарным (он требует наличия двух операндов, по одному с каждой стороны), но первое значение отсутствует, поэтому команда `test` ожидает встретить унарный оператор (такой, как `-z`). Далее, поскольку `test` вернула ненулевой код завершения (из-за ошибки), команда `if` получит ненулевой код завершения, примет соответствующее решение и выполнит вторую команду `echo`.

Эту проблему можно исправить, заключив в кавычки первый аргумент команды `test`:

```
[ "$number" = 1 ]
```

Теперь подстановка приведет к следующему результату:

```
[ "" = 1 ]
```

с правильным числом аргументов. Кавычки следует использовать не только для предохранения от пустых строк, но и в том случае, если переменная содержит строку с несколькими словами, например имя файла со встроенными пробелами.

ПРИМЕЧАНИЕ

Возьмите за правило всегда заключать в кавычки переменные и операции подстановки, когда не требуется разбивать значение на слова.

Логические ошибки

Логические ошибки, в отличие от синтаксических, не прерывают выполнение сценария. Сценарий работает, но желаемых результатов вы не дождетесь, и причина этому — проблемы с логикой. Существует бесчисленное множество возможных логических ошибок, ниже перечислены наиболее типичные их виды, встречающиеся в сценариях:

- **Неправильное условное выражение.** Очень легко неправильно запрограммировать оператор `if/then/else` и получить ошибочную логику работы. Иногда логика получается полностью обратной желаемой или не охватывает весь возможный набор ситуаций.
- **Ошибки «смещения на единицу».** При программировании циклов со счетчиками можно упустить из виду, что цикл должен начинать считать с 0, а не с 1, чтобы счет закончился в нужной точке. Ошибки этого вида приводят к тому, что цикл выполняет на одну итерацию больше или меньше, заканчиваясь соответственно слишком поздно или слишком рано.
- **Непредвиденные ситуации.** Большинство логических ошибок приводят к тому, что программа сталкивается с данными или с ситуацией, не предусмотренными программистом. К ним относятся непредвиденная подстановка, как, например, в случае с именами файлов, содержащими пробелы, которые преобразуются в несколько аргументов команды вместо одного.

Защитное программирование

При программировании важно не опираться на допущения, то есть тщательно проверять коды завершения программ и команд, используемых сценарием. Вот

пример из реальной жизни. Системный горе-администратор написал сценарий, выполняющий некую административную задачу на очень важном сервере. Этот сценарий содержал следующие две строки кода:

```
cd $dir_name
rm *
```

В самих строках нет никакой ошибки, при условии, что каталог, указанный в переменной `dir_name`, действительно существует. Но что случится, если это не так? Тогда команда `cd` потерпит неудачу, сценарий перейдет к следующей строке и удалит файлы в текущем рабочем каталоге. Результат, как вы понимаете, далек от ожидаемого! Несчастный администратор уничтожил массу важных файлов на сервере из-за этой логической ошибки.

Рассмотрим несколько способов усовершенствования описанной логики. Прежде всего можно заставить сценарий развернуть содержимое переменной `dir_name` в одно слово, заключая ее в кавычки, и поставить вызов команды `rm` в зависимость от успеха `cd`:

```
cd "$dir_name" && rm *
```

В этом случае, если команда `cd` потерпит неудачу, команда `rm` не будет выполнена. Так намного лучше, но еще остается вероятность отсутствия переменной `dir_name` или хранения в ней пустого значения, что, безусловно, приведет к удалению файлов в домашнем каталоге пользователя. Этого можно избежать, убедившись, что `dir_name` действительно содержит имя существующего каталога:

```
[[ -d "$dir_name" ]] && cd "$dir_name" && rm *
```

В подобных ситуациях, описанных выше, лучше прервать выполнение сценария с выводом сообщения об ошибке:

```
# Удаление файлов в каталоге $dir_name
if [[ ! -d "$dir_name" ]]; then
      echo "No such directory: '$dir_name'" >&2
      exit 1
fi
if ! cd "$dir_name"; then
      echo "Cannot cd to '$dir_name'" >&2
      exit 1
fi
if ! rm *; then
      echo "File deletion failed. Check results" >&2
      exit 1
fi
```

> **ПЕРЕНОСИМЫЕ ИМЕНА ФАЙЛОВ**
>
> Чтобы гарантировать переносимость имен файлов между разными платформами (например, разными типами компьютеров и операционными системами), ограничьте набор символов, которые можно использовать в именах файлов. Существует стандарт POSIX Portable Filename Character Set (POSIX-совместимый набор символов для переносимых имен файлов), который можно использовать, чтобы увеличить вероятность совместимости имени файла с разными системами. Стандарт очень прост. Он допускает использовать в именах файлов только заглавные *A–Z* и строчные *a–z*, буквы, цифры 0–9, точку (`.`), дефис (`-`) и подчеркивание (_). Также стандарт определяет, что имена файлов не могут начинаться с дефиса.

Здесь проверяются существование каталога с указанным именем и успешное завершение команды `cd`. Если какая-то из проверок завершается неудачей, в стандартный вывод ошибок отправляется содержательное описание и сценарий завершается с кодом 1, чтобы показать, что он завершился с ошибкой.

Будьте внимательны к именам файлов

У этого сценария есть еще одна проблема, неочевидная, но очень опасная. Unix (и Unix-подобные операционные системы) по общему признанию имеет существенный недостаток, касающийся имен файлов, — чрезмерно либеральное отношение к ним. Фактически имена файлов не могут включать только два символа: слеш (`/`), поскольку он используется для разделения элементов путей в файловой системе, и «пустой» символ (с нулевым кодом), который внутренне используется для обозначения концов строк. Все остальные символы считаются допустимыми, включая пробелы, табуляции, переводы строк, ведущие дефисы, возвраты каретки и т. д.

Особую сложность вызывают ведущие дефисы. Например, ничто не помешает создать файл с именем *-rf ~*. А теперь представьте, что случится, если передать имя этого файла команде `rm`.

Чтобы защититься от этой проблемы, нужно заменить команду удаления файлов в сценарии:

```
rm *
```

следующей командой:

```
rm ./*
```

Это предотвратит интерпретацию имен файлов, начинающихся с дефиса, как параметров команды. Возьмите за правило всегда предварять групповые символы (такие, как `*` и `?`) комбинацией `./`, чтобы предотвратить неверную интерпретацию команд. Примером таких подстановок с групповыми символами могут служить `*.pdf` или `???.mp3`.

Проверка ввода

Главное правило надежного программирования: если программа принимает ввод, она должна уметь обработать все, что ей передали. Обычно это означает тщательную отбраковку ввода с целью гарантировать, что дальнейшей обработке будут подвергнуты только допустимые данные. Пример такой проверки мы видели в предыдущей главе, когда обсуждали команду `read`. Там один из сценариев содержал следующую проверку выбранного пункта меню:

```
[[ "$REPLY" =~ ^[0-3]$ ]]
```

Это очень специализированная проверка. Она возвращает код завершения 0, только если строка, введенная пользователем, содержит число в диапазоне от 0 до 3. Никакой другой ввод не принимается. Иногда писать такие проверки очень утомительно, но они совершенно необходимы, если вы хотите в результате получить надежно работающий сценарий.

УДАЧНЫЙ ДИЗАЙН ЕСТЬ ФУНКЦИЯ ОТ ВРЕМЕНИ

Когда я в студенчестве изучал промышленное проектирование, мудрый профессор учил нас, что степень проработки проекта определяется объемом времени, выделенного проектировщику. Если вам дано 5 минут на проектирование устройства для уничтожения воздушных целей, вы спроектируете мухобойку. А если срок — 5 месяцев, вы сможете спроектировать систему противовоздушной обороны с лазерным прицеливанием.

Тот же принцип действует и в программировании. В некоторых случаях допустимо писать сценарии на скорую руку, но только если они будут использоваться один раз и только программистом. Потребность в таких сценариях возникает довольно часто, и они должны разрабатываться быстро, без затраты лишних усилий. Подобные сценарии не требуют подробных комментариев и защитных проверок. С другой стороны, если сценарий предназначен для постоянного использования, то есть он будет использоваться снова и снова для решения важных задач или множеством пользователей, к его разработке следует подходить с большим тщанием.

Тестирование

Тестирование — важный этап в разработке любого программного обеспечения, включая сценарии. В мире открытого программного обеспечения в ходу высказывание «выпускай раньше, выпускай чаще», отражающее этот факт. Программное обеспечение, выпускаемое раньше и чаще, получает больше времени на использование и тестирование. Опыт показывает, что ошибки тем легче найти и тем дешевле исправить, чем раньше в цикле разработки они будут обнаружены.

В главе 26 мы продемонстрировали использование заглушек для проверки потока выполнения программы. Это ценный прием проверки прогресса в работе, начиная с самых ранних стадий разработки сценария.

Вернемся к рассматривавшейся выше задаче удаления файлов и посмотрим, как можно было бы легко протестировать ее решение. Тестировать оригинальный фрагмент довольно опасно, потому что его задача — удаление файлов, но его можно изменить, чтобы сделать тестирование безопасным:

```
if [[ -d $dir_name ]]; then
      if cd $dir_name; then
            echo rm * # ТЕСТИРОВАНИЕ
      else
            echo "cannot cd to '$dir_name'" >&2
            exit 1
      fi
else
      echo "no such directory: '$dir_name'" >&2
      exit 1
fi
exit # ТЕСТИРОВАНИЕ
```

Так как проверка ошибочных условий уже выводит содержательные сообщения, нам не требуется добавлять ничего нового. Самое важное изменение заключается в добавлении команды `echo` перед командой `rm`, которая выведет ее и список ее аргументов, но не разрешит ей выполниться. Это изменение позволит безопасно выполнить код. В конец фрагмента мы добавили команду `exit`, чтобы завершить тест и предотвратить выполнение любых других частей сценария. Необходимость этого шага зависит от предназначения сценария.

Мы также включили несколько комментариев, которые служат «маркерами» изменений, имеющих отношение к тестированию. С их помощью легко можно найти и удалить эти изменения по завершении тестирования.

Комплекты тестов

Чтобы извлечь пользу из тестирования, важно создавать и применять качественные *комплекты тестов*. Для этого следует тщательно подобрать данные для ввода или условия работы, отражающие *крайние* и *пограничные* ситуации. В нашем фрагменте кода (который очень прост) мы хотим проверить, как действует код в трех случаях:

- `dir_name` содержит имя существующего каталога;
- `dir_name` содержит имя несуществующего каталога;
- `dir_name` содержит пустое значение.

Проверив каждое из этих условий, мы получим приличный *охват тестированием*.

Так же как в случае с проектированием, тестирование есть функция от времени. Не каждую особенность сценария нужно тщательно тестировать. В действительности выбор фрагментов для тестирования зависит от того, что считается важным. Поскольку наш фрагмент может нести разрушительные последствия, он заслуживает и тщательного проектирования, и тщательного тестирования.

Отладка

Если тестирование выявляет проблему в сценарии, то наступает черед следующего шага — отладки. Под «проблемой» обычно понимается несоответствие результатов работы сценария ожиданиям программиста. В этом случае нужно точно отследить, что сценарий делает и почему. Поиск ошибок иногда очень напоминает детективное расследование.

Тщательное проектирование сценария может помочь в этом. Согласно принципу защитного программирования, сценарий должен обнаруживать ненормальные условия и выводить содержательные сообщения. Иногда, однако, возникают странные и неожиданные проблемы, требующие применения более сложных приемов защиты.

Поиск проблемной области

В некоторых сценариях, особенно длинных, иногда полезным оказывается использование приема изолирования области сценария, связанной с проблемой. Проблема не всегда является ошибкой, но изоляция часто помогает понять суть происходящего. Один из приемов изоляции заключается в том, чтобы «закомментировать»

фрагмент сценария. Например, попробуем изменить наш фрагмент, удаляющий содержимое каталога, чтобы определить, имеет ли он отношение к ошибке:

```
if [[ -d $dir_name ]]; then
      if cd $dir_name; then
            rm *
      else
            echo "cannot cd to '$dir_name'" >&2
            exit 1
      fi
# else
#       echo "no such directory: '$dir_name'" >&2
#       exit 1
fi
```

Поместив символы комментария в начало каждой строки внутри логического раздела сценария, мы предотвратили возможность выполнения этого раздела. Последующее повторное тестирование покажет, связан ли исключенный код с ошибочным поведением.

Трассировка

Ошибки часто становятся причиной неожиданного направления выполнения сценария. То есть фрагменты сценария могут никогда не выполняться или выполняться в неправильном порядке или в неправильные моменты. Чтобы увидеть, как в действительности протекает выполнение программы, воспользуемся приемом *трассировки*.

Один из способов трассировки заключается в размещении информативных сообщений в разных точках сценария, сообщающих, где протекает выполнение. Например, добавим в наш фрагмент следующие сообщения:

```
echo "preparing to delete files" >&2
if [[ -d $dir_name ]]; then
      if cd $dir_name; then
echo "deleting files" >&2
            rm *
      else
            echo "cannot cd to '$dir_name'" >&2
            exit 1
      fi
else
      echo "no such directory: '$dir_name'" >&2
      exit 1
fi
echo "file deletion complete" >&2
```

Здесь сообщения посылаются в стандартный вывод ошибок, чтобы отделить их от обычного вывода. Кроме того, отсутствуют отступы перед строками с сообщениями — это упростит их поиск, когда придет время убрать эти строки.

Теперь, запустив сценарий, убедимся, что удаление файлов действительно было выполнено:

```
[me@linuxbox ~]$ deletion-script
preparing to delete files
deleting files
file deletion complete
[me@linuxbox ~]$
```

Кроме того, `bash` поддерживает встроенный метод трассировки, реализованный в виде параметра `-x` и команды `set` с параметром `-x`. Возьмем для примера сценарий `trouble`, написанный ранее, и активируем встроенный механизм трассировки для всего сценария, добавив параметр `-x` в первую строку:

```
#!/bin/bash -x

# trouble: сценарий для демонстрации распространенных видов ошибок

number=1

if [ $number = 1 ]; then
      echo "Number is equal to 1."
else
      echo "Number is not equal to 1."
fi
```

После запуска мы получим следующие результаты:

```
[me@linuxbox ~]$ trouble
+ number=1
+ '[' 1 = 1 ']'
+ echo 'Number is equal to 1.'
Number is equal to 1.
```

Включенный механизм трассировки позволяет увидеть, какой вид приобретают команды после применения подстановки. Начальные знаки «плюс» помогают отличить трассировочную информацию от обычного вывода. Знак «плюс» — это символ по умолчанию, используемый для вывода трассировки. Он хранится в переменной командной оболочки `PS4` (prompt string 4 — строка приглашения 4). Изменим значение этой переменной, чтобы сделать трассировочный вывод более полезным. Ниже мы изменили эту переменную, включив в трассировочный вывод текущий номер выполняемой строки в сценарии. Обратите внимание на необходимость использования одиночных кавычек — это

предотвращает подстановку до момента, когда строка приглашения не будет использоваться фактически:

```
[me@linuxbox ~]$ export PS4='$LINENO + '
[me@linuxbox ~]$ trouble
5 + number=1
7 + '[' 1 = 1 ']'
8 + echo 'Number is equal to 1.'
Number is equal to 1.
```

Выполнить трассировку только выбранного фрагмента сценария можно с помощью команды `set` с параметром `-x`:

```
#!/bin/bash

# trouble: сценарий для демонстрации распространенных видов ошибок

number=1

set -x # Включить трассировку
if [ $number = 1 ]; then
      echo "Number is equal to 1."
else
      echo "Number is not equal to 1."
fi
set +x # Выключить трассировку
```

Здесь мы использовали команду `set` с параметром `-x`, чтобы включить трассировку, и с параметром `+x`, чтобы выключить ее. Этот прием используется для исследования сразу нескольких проблемных фрагментов в сценарии.

Исследование значений в процессе выполнения

Часто вместе с трассировкой полезно выводить содержимое переменных, чтобы иметь более полное представление о действиях сценария. Обычно для этого используются дополнительные инструкции `echo`:

```
#!/bin/bash

# trouble: сценарий для демонстрации распространенных видов ошибок

number=1

echo "number=$number" # ОТЛАДКА
set -x # Включить трассировку
```

```
if [ $number = 1 ]; then
      echo "Number is equal to 1."
else
      echo "Number is not equal to 1."
fi
set +x # Выключить трассировку
```

В этом тривиальном примере мы просто вывели значение переменной `number` и отметили дополнительную строку комментарием, чтобы в будущем упростить ее поиск и удаление. Подобный прием особенно полезен при исследовании поведения циклов и арифметических операций в сценариях.

Заключение

В этой главе мы рассмотрели несколько проблем, с которыми можно столкнуться в процессе разработки сценариев. Конечно, таких проблем несоизмеримо больше. Приемы, описанные здесь, помогут вам в поиске наиболее распространенных видов ошибок. Отладка — это искусство предотвращения ошибок (за счет постоянного тестирования в ходе разработки) и их поиска (путем эффективного использования приемов трассировки).

31 Управление потоком выполнения: ветвление с помощью case

В этой главе мы продолжим знакомство с инструментами управления потоком выполнения. В главе 28 мы сконструировали простое меню и реализовали логику обработки выбора его пунктов пользователем. Для этого использовалась серия команд `if`, выясняющих, какой из возможных вариантов выбран. Такие конструкции часто можно увидеть в программах, причем так часто, что в некоторых языках программирования (включая командную оболочку) был реализован механизм управления потоком выполнения для случаев с множеством альтернативных вариантов.

Команда case

Командная оболочка `bash` поддерживает составную команду выбора из нескольких вариантов, которая называется `case`. Она имеет следующий синтаксис:

```
case слово in
      [шаблон [| шаблон]...) команды ;;]...
esac
```

Взгляните еще раз, как программа `read-menu` из главы 28 обрабатывает выбор пользователя:

```
#!/bin/bash

# read-menu: программа вывода системной информации,
#            управляемая с помощью меню
```

```
clear
echo "
Please Select:

1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"
read -p "Enter selection [0-3] > "

if [[ "$REPLY" =~ ^[0-3]$ ]]; then
      if [[ "$REPLY" == 0 ]]; then
            echo "Program terminated."
            exit
      fi
      if [[ "$REPLY" == 1 ]]; then
            echo "Hostname: $HOSTNAME"
            uptime
            exit
      fi
      if [[ "$REPLY" == 2 ]]; then
            df -h
            exit
      fi
      if [[ "$REPLY" == 3 ]]; then
            if [[ "$(id -u)" -eq 0 ]]; then
                  echo "Home Space Utilization (All Users)"
                  du -sh /home/*
            else
                  echo "Home Space Utilization ($USER)"
                  du -sh "$HOME"
            fi
            exit
      fi
else
      echo "Invalid entry." >&2
      exit 1
fi
```

С помощью `case` можно сделать логику выбора немного проще:

```
#!/bin/bash

# case-menu: программа вывода системной информации,
```

```
#           управляемая с помощью меню

clear
echo "
Please Select:

1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"
read -p "Enter selection [0-3] > "

case "$REPLY" in
    0)    echo "Program terminated."
          exit
          ;;
    1)    echo "Hostname: $HOSTNAME"
          uptime
          ;;
    2)    df -h
          ;;
    3)    if [[ "$(id -u)" -eq 0 ]]; then
              echo "Home Space Utilization (All Users)"
              du -sh /home/*
          else
              echo "Home Space Utilization ($USER)"
              du -sh "$HOME"
          fi
          ;;
    *)    echo "Invalid entry" >&2
          exit 1
          ;;
esac
```

Команда `case` берет значение *слова* — в данном примере значение переменной `REPLY` — и затем сопоставляет его с указанными *шаблонами*. Найдя соответствие, она выполняет команды, связанные с найденным шаблоном. После нахождения соответствия сопоставление с нижележащими шаблонами уже не производится.

Шаблоны

Шаблоны обрабатываются командой `case` точно так же, как пути механизмом подстановки. Шаблоны завершаются символом `)`. В табл. 31.1 перечислены некоторые допустимые шаблоны.

Таблица 31.1. Примеры шаблонов в команде case

Шаблон	Описание
`a)`	Соответствует, если *слово* содержит a
`[[:alpha:]])`	Соответствует, если *слово* содержит единственный алфавитный символ
`???)`	Соответствует, если *слово* содержит ровно три символа
`*.txt)`	Соответствует, если *слово* заканчивается символами `.txt`
`*)`	Соответствует любому значению *слова*. Считается хорошей практикой включать этот шаблон в команду `case` последним, чтобы перехватывать любые значения *слова*, не соответствующие ни одному из предыдущих шаблонов, то есть чтобы перехватывать любые недопустимые значения

Следующий пример демонстрирует работу шаблонов:

```
#!/bin/bash

read -p "enter word > "

case "$REPLY" in
    [[:alpha:]])   echo "is a single alphabetic character." ;;
    [ABC][0-9])    echo "is A, B, or C followed by a digit." ;;
    ???)           echo "is three characters long." ;;
    *.txt)         echo "is a word ending in '.txt'" ;;
    *)             echo "is something else." ;;
esac
```

Мы можем объединить несколько шаблонов, перечислив их через символ вертикальной черты. В результате получается комбинированный условный шаблон, объединенный по «ИЛИ». Эта возможность может пригодиться, например, для обработки символов верхнего и нижнего регистров:

```
#!/bin/bash

# case-menu: программа вывода системной информации,
#            управляемая с помощью меню

clear
echo "
Please Select:
A. Display System Information
```

```
B. Display Disk Space
C. Display Home Space Utilization
Q. Quit
"
read -p "Enter selection [A, B, C or Q] > "

case "$REPLY" in
    q|Q)    echo "Program terminated."
            exit
            ;;
    a|A)    echo "Hostname: $HOSTNAME"
            uptime
            ;;
    b|B)    df -h
            ;;
    c|C)    if [[ "$(id -u)" -eq 0 ]]; then
                echo "Home Space Utilization (All Users)"
                du -sh /home/*
            else
                echo "Home Space Utilization ($USER)"
                du -sh "$HOME"
            fi
            ;;
    *)      echo "Invalid entry" >&2
            exit 1
            ;;
esac
```

Здесь мы изменили программу `case-menu`, предложив пользователю выбирать пункты меню вводом букв, а не цифр. Обратите внимание, что новые шаблоны позволяют вводить буквы обоих регистров — верхнего и нижнего.

Выполнение нескольких вариантов

В версиях `bash` до 4.0 команда `case` могла выполнить только один вариант, соответствующий совпавшему шаблону. После этого команда завершалась. Рассмотрим сценарий, проверяющий введенный символ:

```
#!/bin/bash

# case4-1: проверка символа

read -n 1 -p "Type a character > "
echo
case "$REPLY" in
```

```
    [[:upper:]])   echo "'$REPLY' is upper case." ;;
    [[:lower:]])   echo "'$REPLY' is lower case." ;;
    [[:alpha:]])   echo "'$REPLY' is alphabetic." ;;
    [[:digit:]])   echo "'$REPLY' is a digit." ;;
    [[:graph:]])   echo "'$REPLY' is a visible character." ;;
    [[:punct:]])   echo "'$REPLY' is a punctuation symbol." ;;
    [[:space:]])   echo "'$REPLY' is a whitespace character." ;;
    [[:xdigit:]])  echo "'$REPLY' is a hexadecimal digit." ;;
esac
```

Вот как выглядит результат выполнения этого сценария:

```
[me@linuxbox ~]$ case4-1
Type a character > a
'a' is lower case.
```

В большинстве случаев сценарий прекрасно справляется со своей задачей, но терпит неудачу, если символ соответствует нескольким символьным классам POSIX. Например, символ `a` соответствует классам алфавитных символов и символов нижнего регистра, а также шестнадцатеричных цифр. В `bash` до версии 4.0 не было никакой возможности заставить `case` выполнить больше одной успешной проверки. Современные версии `bash` поддерживают дополнительную нотацию `;;&` в конце каждого варианта, которая используется, как показано ниже:

```
#!/bin/bash

# case4-2: проверка символа

read -n 1 -p "Type a character > "
echo
case "$REPLY" in
    [[:upper:]])   echo "'$REPLY' is upper case." ;;&
    [[:lower:]])   echo "'$REPLY' is lower case." ;;&
    [[:alpha:]])   echo "'$REPLY' is alphabetic." ;;&
    [[:digit:]])   echo "'$REPLY' is a digit." ;;&
    [[:graph:]])   echo "'$REPLY' is a visible character." ;;&
    [[:punct:]])   echo "'$REPLY' is a punctuation symbol." ;;&
    [[:space:]])   echo "'$REPLY' is a whitespace character." ;;&
    [[:xdigit:]])  echo "'$REPLY' is a hexadecimal digit." ;;&
esac
```

Запустив этот сценарий, мы получим:

```
[me@linuxbox ~]$ case4-2
Type a character > a
'a' is lower case.
```

```
'a' is alphabetic.
'a' is a visible character.
'a' is a hexadecimal digit.
```

Дополнительный синтаксис `;;&` позволяет команде `case` продолжить проверку вместо простого завершения после первого найденного совпадения.

Заключение

Команда `case` является удобным дополнением к нашей коллекции приемов программирования. Как будет показано в следующей главе, она отлично подходит для решения некоторых видов задач.

32 Позиционные параметры

Во всех предыдущих наших программах отсутствовала одна особенность — возможность принимать и обрабатывать параметры и аргументы командной строки. В этой главе мы исследуем эту возможность и позволим нашим программам обращаться к содержимому командной строки.

Доступ к командной строке

Командная оболочка поддерживает множество переменных, которые называются *позиционными параметрами* и содержат отдельные слова из командной строки. Эти переменные имеют имена от `0` до `9`. Продемонстрируем их:

```
#!/bin/bash

# posit-param: сценарий для просмотра параметров командной строки

echo "
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = $3
\$4 = $4
\$5 = $5
\$6 = $6
\$7 = $7
\$8 = $8
\$9 = $9
"
```

Этот очень простой сценарий выводит значения переменных с именами от `$0` до `$9`. Запустим его без аргументов командной строки:

```
[me@linuxbox ~]$ posit-param

$0 = /home/me/bin/posit-param
$1 =
$2 =
$3 =
$4 =
$5 =
$6 =
$7 =
$8 =
$9 =
```

Даже в отсутствие аргументов переменная `$0` всегда содержит первый элемент командной строки — путь к файлу выполняемой программы. Давайте передадим сценарию несколько аргументов:

```
[me@linuxbox ~]$ posit-param a b c d

$0 = /home/me/bin/posit-param
$1 = a
$2 = b
$3 = c
$4 = d
$5 =
$6 =
$7 =
$8 =
$9 =
```

ПРИМЕЧАНИЕ

В действительности, если использовать механизм подстановки параметров, можно получить доступ более чем к девяти параметрам. Чтобы указать число больше девяти, следует заключить его в фигурные скобки; например, `${10}`, `${55}`, `${211}` и т. д.

Определение числа аргументов

Командная оболочка поддерживает также переменную `$#`, хранящую число аргументов командной строки:

```
#!/bin/bash

# posit-param: сценарий для просмотра параметров командной строки

echo "
Number of arguments: $#
\$0 = $0
\$1 = $1
\$2 = $2
\$3 = $3
\$4 = $4
\$5 = $5
\$6 = $6
\$7 = $7
\$8 = $8
\$9 = $9
"
```

Результат:

```
[me@linuxbox ~]$ posit-param a b c d

Number of arguments: 4
$0 = /home/me/bin/posit-param
$1 = a
$2 = b
$3 = c
$4 = d
$5 =
$6 =
$7 =
$8 =
$9 =
```

shift — доступ к множеству аргументов

Но как быть, если программе передается большое число аргументов, как в следующем примере:

```
[me@linuxbox ~]$ posit-param *

Number of arguments: 82
$0 = /home/me/bin/posit-param
$1 = addresses.ldif
$2 = bin
$3 = bookmarks.html
```

```
$4 = debian-500-i386-netinst.iso
$5 = debian-500-i386-netinst.jigdo
$6 = debian-500-i386-netinst.template
$7 = debian-cd_info.tar.gz
$8 = Desktop
$9 = dirlist-bin.txt
```

В системе, где выполнялся этот пример, механизм подстановки развернул символ * в 82 аргумента. Как обработать такое количество? Командная оболочка предусматривает решение и для подобных случаев, правда, следует отметить, что изяществом оно не отличается. Команда `shift` выполняет «сдвиг» параметров к началу списка. Фактически, используя `shift`, можно обойтись единственной переменной-параметром (помимо `$0`, которая никогда не изменяется).

```
#!/bin/bash

# posit-param2: сценарий вывода всех аргументов

count=1

while [[ $# -gt 0 ]]; do
      echo "Argument $count = $1"
      count=$((count + 1))
      shift
done
```

Каждый раз, когда выполняется команда `shift`, значение `$2` перемещается в `$1`, значение `$3` перемещается в `$2` и т. д. Значение `$#` при этом уменьшается на 1.

В программе `posit-param2` мы создали цикл, проверяющий число оставшихся аргументов и продолжающийся до тех пор, пока оно не уменьшится до нуля. Цикл выводит текущий аргумент, в каждой итерации увеличивает счетчик обработанных аргументов `count` и, наконец, выполняет `shift`, чтобы загрузить в `$1` следующий аргумент. Вот как работает эта программа:

```
[me@linuxbox ~]$ posit-param2 a b c d
Argument 1 = a
Argument 2 = b
Argument 3 = c
Argument 4 = d
```

Простые приложения

Даже без команды `shift` можно писать полезные приложения, использующие позиционные параметры. Например, ниже приводится простая программа получения информации о файле:

```
#!/bin/bash

# file_info: простая программа получения информации о файле

PROGNAME="$(basename "$0")"

if [[ -e "$1" ]]; then
      echo -e "\nFile Type:"
      file "$1"
      echo -e "\nFile Status:"
      stat "$1"
else
      echo "$PROGNAME: usage: $PROGNAME file" >&2
      exit 1
fi
```

Эта программа выводит тип указанного файла (определяется с помощью команды `file`) и его атрибуты (командой `stat`). Интересной особенностью программы является переменная `PROGNAME`. Ей присваивается результат выполнения команды `basename "$0"`. Команда `basename` удаляет начальную часть из пути к файлу, оставляя только базовое имя. В данном примере `basename` удалит начальную часть из параметра `$0`, хранящего полный путь к данной программе. Такой результат удобно использовать для конструирования сообщений, например, о правилах использования программы. При подобном подходе можно переименовать сценарий, и при выводе сообщений новое имя программы будет использоваться автоматически.

Использование позиционных параметров в функциях

Позиционные параметры используются для передачи аргументов не только в сценарии, но и в функции командной оболочки. Для демонстрации преобразуем сценарий `file_info` в функцию:

```
file_info () {

      # file_info: функция для вывода информации о файле

      if [[ -e "$1" ]]; then
            echo -e "\nFile Type:"
            file "$1"
            echo -e "\nFile Status:"
            stat "$1"
      else
            echo "$FUNCNAME: usage: $FUNCNAME file" >&2
            return 1
      fi
}
```

Теперь, если сценарий, включающий функцию `file_info`, вызовет ее с именем файла в аргументе, аргумент будет передан в функцию.

Благодаря этому мы получаем возможность написать множество полезных функций для использования не только в наших сценариях, но и в файле *.bashrc*.

Обратите внимание, что в этом примере вместо переменной `PROGNAME` используется переменная командной оболочки `FUNCNAME`. Оболочка автоматически присваивает значение этой переменной в момент вызова функции. Отметьте также, что `$0` всегда содержит полный путь к первому элементу командной строки (то есть имя программы), а не имя функции, как можно было бы ожидать.

Обработка позиционных параметров скопом

Иногда бывает необходимо выполнить операцию сразу со всеми позиционными параметрами. Например, может понадобиться написать «обертку» для некоторой программы, то есть сценарий или функцию, упрощающие запуск этой программы. Обертка принимает список непонятных для нее параметров командной строки и просто передает его обертываемой программе.

Для этой цели командная оболочка предоставляет два специальных параметра. Они оба замещаются полным списком позиционных параметров, но имеют некоторые тонкие отличия. Описание этих параметров приводится в табл. 32.1.

Таблица 32.1. Специальные параметры $* и $@

Параметр	Описание
`$*`	Замещается списком позиционных параметров, начиная с `$1`. Если имя параметра `$*` заключить в двойные кавычки, позиционные параметры будут перечислены в списке через первый символ в переменной `IFS` (по умолчанию пробел), а сам список будет размещен в одной строке и заключен в кавычки
`$@`	Замещается списком позиционных параметров, начиная с `$1`. Если имя параметра `$@` заключить в двойные кавычки, механизм подстановки заменит его списком позиционных параметров, заключенных в кавычки по отдельности

Следующий сценарий демонстрирует, как действуют эти специальные параметры:

```
#!/bin/bash

# posit-params3 : сценарий для демонстрации $* и $@
```

```
print_params () {
      echo "\$1 = $1"
      echo "\$2 = $2"
      echo "\$3 = $3"
      echo "\$4 = $4"
}

pass_params () {
      echo -e "\n" '$* :';   print_params $*
      echo -e "\n" '"$*" :'; print_params "$*"
      echo -e "\n" '$@ :';   print_params $@
      echo -e "\n" '"$@" :'; print_params "$@"
}

pass_params "word" "words with spaces"
```

В этой довольно замысловатой программе мы создали два аргумента, `word` и `words with spaces`, и передали их функции `pass_params`. Эта функция, в свою очередь, передает их функции `print_params`, с применением каждого из четырех методов, доступных для специальных параметров `$*` и `$@`. Вывод сценария показывает разницу между ними:

```
[me@linuxbox ~]$ posit-param3
 $* :
$1 = word
$2 = words
$3 = with
$4 = spaces

 "$*" :
$1 = word words with spaces
$2 =
$3 =
$4 =

 $@ :
$1 = word
$2 = words
$3 = with
$4 = spaces

 "$@" :
$1 = word
$2 = words with spaces
$3 =
$4 =
```

В данном примере оба параметра, `$*` и `$@`, возвращают результат из четырех слов:

```
word words with spaces
```

`"$*"` возвращает результат в виде одного слова, содержащего пробелы:

```
"word words with spaces"
```

`"$@"` возвращает результат в виде двух слов, второе из которых включает пробелы:

```
"word" "words with spaces"
```

Это соответствует нашим фактическим намерениям. Этот пример показывает, что, несмотря на наличие четырех разных способов получения списка позиционных параметров, в большинстве ситуаций предпочтительнее использовать прием с `"$@"`, потому что он сохраняет целостность каждого позиционного параметра.

Более сложное приложение

После долгой паузы мы продолжим работу над программой `sys_info_page`. Теперь мы добавим в нее поддержку нескольких параметров командной строки:

- **Выходной файл.** Мы добавим параметр, который позволит указать имя файла для вывода результатов работы программы. Сделать это можно будет с помощью `-f файл` или `--file файл`.
- **Интерактивный режим.** При передаче этого параметра программа будет предлагать пользователю ввести имя выходного файла и определять, существует ли этот файл. Если файл существует, пользователю будет предложено подтвердить свое решение, прежде чем затереть существующий файл. Этот параметр можно будет передать как `-i` или `--interactive`.
- **Справка.** Передав параметр `-h` или `--help`, можно потребовать от программы вывести сообщение с информацией о правилах пользования программой.

Далее приводится код, реализующий обработку командной строки:

```
usage () {
      echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"
      return
}

# обработка параметров командной строки

interactive=
filename=
```

```
while [[ -n "$1" ]]; do
      case "$1" in
            -f | --file)           shift
                                   filename="$1"
                                   ;;
            -i | --interactive)    interactive=1
                                   ;;
            -h | --help)           usage
                                   exit
                                   ;;
            *)                     usage >&2
                                   exit 1
                                   ;;
      esac
      shift
done
```

Сначала мы добавили функцию `usage` для вывода сообщения, если программа вызывается с параметром `--help` или с неизвестным параметром.

Затем следует цикл обработки параметров. Цикл продолжается, пока позиционный параметр `$1` не получит пустое значение. В конце цикла вызывается команда `shift`, чтобы сдвинуть позиционные параметры и, в конечном итоге, гарантировать завершение цикла.

Внутри цикла инструкция `case` проверяет текущий позиционный параметр на соответствие поддерживаемым вариантам. Если данный параметр поддерживается, выполняется соответствующая операция, если нет — выводится сообщение с информацией о правилах пользования программой и сценарий завершается с признаком ошибки.

Обратите внимание, как обрабатывается параметр `-f`. Обнаружив этот параметр, программа выполняет команду `shift`, которая сдвинет аргумент параметра `-f` с именем файла в позиционный параметр `$1`.

Далее следует код, реализующий интерактивный режим:

```
# интерактивный режим

if [[ -n "$interactive" ]]; then
      while true; do
            read -p "Enter name of output file: " filename
            if [[ -e "$filename" ]]; then
                  read -p "'$filename' exists. Overwrite? [y/n/q] > "
                  case "$REPLY" in
                      Y|y)      break
                                ;;
```

```
                        Q|q)      echo "Program terminated."
                                  exit
                                  ;;
                        *)        continue
                                  ;;
                    esac
              elif [[ -z "$filename" ]]; then
                    continue
              else
                    break
              fi
      done
fi
```

Если переменная `interactive` содержит непустое значение, начинается бесконечный цикл, который предлагает ввести имя файла и затем обрабатывает ситуацию, если введенное имя соответствует существующему файлу. Если указанный файл уже существует, пользователю на выбор предлагается три варианта: затереть существующий файл, выбрать другое имя или завершить программу. Если пользователь предпочтет затереть существующий файл, выполняется команда `break` и цикл прерывается. Обратите внимание, что инструкция `case` различает только вариант перезаписи существующего файла и завершения программы. Любой другой ответ пользователя будет приводить к переходу в начало цикла с повторным предложением ввести имя файла.

Для поддержки вывода в файл сначала необходимо имеющийся код вывода страницы преобразовать в функцию. Необходимость такого решения станет понятна чуть позже:

```
write_html_page () {
      cat <<- _EOF_
      <html>
              <head>
                     <title>$TITLE</title>
              </head>
              <body>
                     <h1>$TITLE</h1>
                     <p>$TIMESTAMP</p>
                     $(report_uptime)
                     $(report_disk_space)
                     $(report_home_space)
              </body>
       </html>
       _EOF_
       return
}
```

```
# вывод страницы html

if [[ -n "$filename" ]]; then
      if touch "$filename" && [[ -f "$filename" ]]; then
            write_html_page > "$filename"
      else
            echo "$PROGNAME: Cannot write file '$filename'" >&2
            exit 1
      fi
else
      write_html_page
fi
```

Код, обслуживающий логику параметра `-f`, находится в конце листинга, приведенного выше. Он проверяет, определено ли имя файла и затем — доступность для записи файла с указанным именем. Для этого выполняется команда `touch` с последующей проверкой, что файл является обычным файлом. Эти две проверки позволяют обработать ситуацию неправильно указанного пути (в этом случае `touch` потерпит неудачу) и убедиться, что существующий файл является обычным файлом.

Как видите, функция `write_html_page` вызывается, чтобы сгенерировать фактическое содержимое страницы, которое затем либо выводится в стандартный вывод (если переменная `filename` содержит пустое значение), либо перенаправляется в указанный файл. Поскольку код HTML может выводиться в файл или в стандартный вывод, есть смысл преобразовать процедуру `write_html_page` в функцию, чтобы избежать повторения кода.

Заключение

С помощью дополнительных позиционных параметров мы можем теперь писать довольно функциональные сценарии. Позиционные параметры помогают создавать очень полезные функции командной оболочки для выполнения повседневных задач, которые можно поместить в файл *.bashrc*.

Наша программа `sys_info_page` выросла и усложнилась. Ниже приводится полный листинг программы с выделенными последними изменениями:

```
#!/bin/bash

# sys_info_page: программа вывода страницы с информацией о системе

PROGNAME="$(basename $0)"
TITLE="System Information Report For $HOSTNAME"
```

```
CURRENT_TIME=$(date +"%x %r %Z")
TIME_STAMP="Generated $CURRENT_TIME, by $USER"

report_uptime () {
      cat <<- _EOF_
            <h2>System Uptime</h2>
            <pre>$(uptime)</pre>
            _EOF_
      return
}

report_disk_space () {
      cat <<- _EOF_
            <h2>Disk Space Utilization</h2>
            <pre>$(df -h)</pre>
            _EOF_
      return
}

report_home_space () {
      if [[ "$(id -u)" -eq 0 ]]; then
            cat <<- _EOF_
                  <h2>Home Space Utilization (All Users)</h2>
                  <pre>$(du -sh /home/*)</pre>
                  _EOF_
      else
            cat <<- _EOF_
                  <h2>Home Space Utilization ($USER)</h2>
                  <pre>$(du -sh "$HOME")</pre>
                  _EOF_
      fi
      return
}

usage () {
      echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"
      return
}

write_html_page () {
      cat <<- _EOF_
      <html>
              <head>
                     <title>$TITLE</title>
              </head>
              <body>
                     <h1>$TITLE</h1>
```

```
                    <p>$TIMESTAMP</p>
                    $(report_uptime)
                    $(report_disk_space)
                    $(report_home_space)
             </body>
       </html>
       _EOF_
       return
}

# обработка параметров командной строки

interactive=
filename=
while [[ -n "$1" ]]; do
    case "$1" in
        -f | --file)           shift
                               filename="$1"
                               ;;
        -i | --interactive)    interactive=1
                               ;;
        -h | --help)           usage
                               exit
                               ;;
        *)                     usage >&2
                               exit 1
                                ;;
    esac
    shift
done

# интерактивный режим

if [[ -n "$interactive" ]]; then
    while true; do
        read -p "Enter name of output file: " filename
        if [[ -e "$filename" ]]; then
            read -p "'$filename' exists. Overwrite? [y/n/q] > "
            case "$REPLY" in
                Y|y)    break
                        ;;
                Q|q)    echo "Program terminated."
                        exit
                        ;;
                *)      continue
                        ;;
            esac
```

```
        fi
    done
fi

# вывод страницы html

if [[ -n "$filename" ]]; then
      if touch "$filename" && [[ -f "$filename" ]]; then
            write_html_page > "$filename"
      else
            echo "$PROGNAME: Cannot write file '$filename'" >&2
            exit 1
      fi
else
      write_html_page
fi
```

У нас уже получился неплохой сценарий, но он еще не закончен. В следующей главе мы добавим в него последнее улучшение.

33 Управление потоком выполнения: цикл for

В этой заключительной главе, посвященной управлению потоком выполнения, мы познакомимся еще с одной конструкцией организации циклов в командной оболочке. *Цикл for* отличается от циклов `while` и `until` поддержкой средств обработки последовательностей. Это очень полезная возможность. Как следствие, цикл `for` пользуется большой популярностью среди создателей сценариев для `bash`.

Цикл `for` реализован, что вполне естественно, в виде команды `for`. В современных версиях `bash` поддерживается две формы команды `for`.

for: традиционная форма

Оригинальный синтаксис команды `for` имеет следующий вид:

```
for переменная [in слова]; do
    команды
done
```

где *переменная* — это имя переменной, значение которой будет увеличиваться в ходе выполнения цикла, `слова` — необязательный список элементов, которые последовательно будут присваиваться переменной, и *команды* — это команды, выполняемые в каждой итерации.

Команду `for` удобно использовать в командной строке. Рассмотрим, как она работает:

```
[me@linuxbox ~]$ for i in A B C D; do echo $i; done
A
B
C
D
```

В этом примере команда `for` получает список из четырех слов: *A*, *B*, *C* и *D*. Для обхода этого списка выполняется четыре итерации цикла. В начале каждой итерации переменной `i` присваивается очередное слово. Внутри цикла находится команда `echo`, она выводит значение `i`, чтобы показать, что присваивание действительно выполняется. Так же как в случае с циклами `while` и `until`, цикл `for` заканчивается ключевым словом `done`.

По-настоящему мощной особенностью `for` является разнообразие способов формирования списка слов. Например, можно использовать подстановку в фигурных скобках:

```
[me@linuxbox ~]$ for i in {A..D}; do echo $i; done
A
B
C
D
```

или подстановку имен файлов:

```
[me@linuxbox ~]$ for i in distros*.txt; do echo "$i"; done
distros-by-date.txt
distros-dates.txt
distros-key-names.txt
distros-key-vernums.txt
distros-names.txt
distros.txt
distros-vernums.txt
distros-versions.txt
```

Механизм подстановки имен файлов позволяет получить четкий список с именами файлов, который можно обработать в цикле. Единственное, о чем следует позаботиться, — убедиться, что механизм подстановки вернул действительные совпадения. По умолчанию, если операция подстановки не найдет ни одного файла, соответствующего шаблону, она вернет сам шаблон (в данном случае `distros*.txt`). Чтобы защититься от этой проблемы, предыдущий код можно преобразовать в сценарий, как показано ниже:

```
for i in distros*.txt; do
      if [[ -e "$i" ]]; then
            echo "$i"
      fi
done
```

Добавив проверку существования файла, мы обезопасили себя от ситуации, когда механизм подстановки не нашел ни одного файла. Другой распространенный метод получения списка слов — подстановка команд.

```
#!/bin/bash

# longest-word : поиск самой длинной строки в файле

while [[ -n "$1" ]]; do
      if [[ -r "$1" ]]; then
            max_word=
            max_len=0
            for i in $(strings "$1"); do
                  len="$(echo "$i" | wc -c)"
                  if (( len > max_len )); then
                        max_len="$len"
                        max_word="$i"
                  fi
            done
            echo "$1: '$max_word' ($max_len characters)"
      fi
      shift
done
```

Этот пример осуществляет поиск самой длинной строки в файле. Когда в командной строке указано несколько имен файлов, сценарий вызывает процедуру `strings` (входит в состав пакета GNU binutils), чтобы получить список «слов» из каждого файла. Цикл `for` обрабатывает каждое слово по очереди и определяет, является ли оно самым длинным из встречавшихся до сих пор. По завершении цикла сценарий выводит самое длинное слово.

Обратите внимание, что здесь, вопреки обычной практике, мы не заключили подстановку команд `$(strings "$1")` в кавычки. Объясняется это просто: здесь мы действительно хотим разбить строку на слова, чтобы получить список. Заключив подстановку команд в кавычки, мы получили бы одно слово, содержащее все строки из файла. А это не совсем то, что нам нужно.

Если необязательный компонент `слова` в команде `for` отсутствует, она по умолчанию обрабатывает позиционные параметры. Чтобы показать использование этого способа, изменим сценарий `longest-word`:

```
#!/bin/bash

# longest-word2 : поиск самой длинной строки в файле

for i; do
      if [[ -r "$i" ]]; then
            max_word=
            max_len=0
            for j in $(strings "$i"); do
```

```
                len="$(echo "$j" | wc -c)"
                if (( len > max_len )); then
                      max_len="$len"
                      max_word="$j"
                fi
          done
          echo "$i: '$max_word' ($max_len characters)"
      fi
done
```

Как видите, мы заменили внешний цикл `while` циклом `for`. Так как список слов в команде `for` отсутствует, она перебирает позиционные параметры. Во внутреннем цикле вместо переменной `i` теперь используется переменная `j`. Кроме того, нам больше не нужна команда `shift`.

ПОЧЕМУ I?

Вы могли заметить, что во всех примерах цикла `for` выше использовалась переменная `i`. Почему? В действительности за этим выбором не стоят какие-то определенные причины, кроме стремления следовать традициям. В команде `for` можно использовать любую допустимую переменную, но чаще всего используется переменная `i`, а также `j` и `k`.

Своими корнями эта традиция уходит в язык программирования Fortran. В Fortran необъявленные переменные, начинающиеся с букв *I*, *J*, *K*, *L* и *M*, автоматически становились целочисленными, тогда как переменные, начинающиеся с любой другой буквы, — действительными, или вещественными (способными хранить числа с дробной частью). Эта особенность вынуждала программистов использовать переменные *I*, *J* и *K* в качестве переменных цикла, так как использование их в качестве временных переменных (чем переменные цикла в действительности и являются) требовало меньших усилий.

Из-за этого даже в среде программистов на Fortran ходила острота: «GOD is real, unless declared integer» (Бог действителен, пока явно не объявлен целым).

for: форма в стиле языка C

В последние версии `bash` добавлена вторая форма синтаксиса команды `for`, напоминающая одноименный оператор в языке программирования C, которая поддерживается также многими другими языками.

```
for (( выражение1; выражение2; выражение3 )); do
      команды
done
```

где *выражение1*, *выражение2* и *выражение3* — это арифметические выражения, а *команды* — это команды, выполняемые в каждой итерации цикла.

Своим поведением эта форма эквивалентна следующей конструкции:

```
(( выражение1 ))
while (( выражение2 )); do
      команды
      (( выражение3 ))
done
```

выражение1 инициализирует цикл, *выражение2* определяет условие завершения цикла, *выражение3* выполняется в конце каждой итерации.

Ниже приводится пример типичного применения:

```
#!/bin/bash

# simple_counter : демонстрация команды for в стиле языка C

for (( i=0; i<5; i=i+1 )); do
      echo $i
done
```

Этот сценарий произведет следующий вывод:

```
[me@linuxbox ~]$ simple_counter
0
1
2
3
4
```

Здесь *выражение1* инициализирует переменную `i` значением 0, `выражение2` позволяет продолжать итерации, пока значение `i` остается меньше 5, `выражение3` увеличивает на единицу значение `i` в конце каждой итерации.

Форма команды `for` в стиле языка C выглядит предпочтительнее, если требуется работать с числовыми последовательностями. Несколько примеров ее применения будут приведены в следующих двух главах.

Заключение

Познакомившись с командой `for`, внесем заключительное усовершенствование в наш сценарий `sys_info_page`. В настоящий момент функция `report_home_space` выглядит так:

```
report_home_space () {
      if [[ "$(id -u)" -eq 0 ]]; then
            cat <<- _EOF_
                  <h2>Home Space Utilization (All Users)</h2>
                  <pre>$(du -sh /home/*)</pre>
                  _EOF_
      else
            cat <<- _EOF_
                  <h2>Home Space Utilization ($USER)</h2>
                  <pre>$(du -sh "$HOME")</pre>
                  _EOF_
      fi
      return
}
```

Теперь мы можем переписать ее, добавив вывод информации о домашнем каталоге каждого пользователя и включив в вывод общее число файлов и подкаталогов в каждом из них:

```
report_home_space () {

      local format="%8s%10s%10s\n"
      local i dir_list total_files total_dirs total_size user_name

      if [[ "$(id -u)" -eq 0 ]]; then
            dir_list=/home/*
            user_name="All Users"
      else
            dir_list="$HOME"
            user_name="$USER"
      fi

      echo "<h2>Home Space Utilization ($user_name)</h2>"

      for i in $dir_list; do

            total_files="$(find "$i" -type f | wc -l)"
            total_dirs="$(find "$i" -type d | wc -l)"
            total_size="$(du -sh "$i" | cut -f 1)"
```

```
        echo "<h3>$i</h3>"
        echo "<pre>"
        printf "$format" "Dirs" "Files" "Size"
        printf "$format" "----" "-----" "----"
        printf "$format" "$total_dirs" "$total_files" "$total_size"
        echo "</pre>"
    done
    return
}
```

В этой новой версии применено многое из того, что мы узнали к данному моменту. Она все еще проверяет наличие привилегий суперпользователя, но вместо того, чтобы выполнить полный набор операций в каждой из ветвей `if`, здесь устанавливаются некоторые переменные, которые затем используются в цикле `for`. В функции использованы несколько локальных переменных и команда `printf` для форматирования части вывода.

34 Строки и числа

Любые компьютерные программы обрабатывают данные. В предыдущих главах основное внимание уделялось обработке данных на уровне файлов. Однако многие задачи решаются с использованием меньших единиц данных, таких как строки и числа.

В этой главе мы рассмотрим некоторые возможности командной оболочки для работы со строками и числами. Командная оболочка поддерживает большое разнообразие способов подстановки параметров, которые выполняют строковые операции. В дополнение к подстановке результатов арифметических выражений (о которой рассказывалось в главе 7) существует программа командной строки `bc`, выполняющая математические операции.

Подстановка параметров

Механизм подстановки параметров уже рассматривался в главе 7, но там этот механизм не был описан детально, потому что большая часть его возможностей используется в сценариях, а не в командной строке. Мы уже знакомы с некоторыми формами подстановки параметров, например, с подстановкой значений переменных командной оболочки. Но в командной оболочке их намного больше.

ПРИМЕЧАНИЕ

Старайтесь всегда заключать операции подстановки параметров в двойные кавычки, чтобы предотвратить нежелательное разделение строк на слова, если для этого нет веских причин. Это особенно актуально при работе с именами файлов, потому что они могут включать пробелы и другие неожиданные символы.

Простые параметры

Простейшую форму подстановки параметров можно наблюдать в использовании переменных. Например, запись

```
$a
```

после подстановки превращается в содержимое переменной a. Простые параметры можно заключать в фигурные скобки, например:

```
${a}
```

Это не оказывает влияния на результат подстановки, но является необходимым, если сразу за именем переменной следует какой-то другой текст, который может сбивать с толку командную оболочку. В следующем примере выполняется попытка сконструировать имя файла добавлением строки _file к содержимому переменной a:

```
[me@linuxbox ~]$ a="foo"
[me@linuxbox ~]$ echo "$a_file"
```

Если выполнить эту последовательность команд, результатом будет пустое значение, потому что командная оболочка попытается выполнить подстановку значения переменной a_file вместо a. Эта проблема устраняется с помощью фигурных скобок:

```
[me@linuxbox ~]$ echo "${a}_file"
foo_file
```

Мы видели также, что доступ к позиционным параметрам с порядковыми номерами выше 9 тоже осуществляется с помощью фигурных скобок. Например, прочитать 11-й позиционный параметр можно следующим образом:

```
${11}
```

Подстановка пустых переменных

Некоторые формы подстановки параметров помогают решать проблемы с несуществующими, или пустыми, переменными. Эти формы удобно использовать для обработки ситуаций отсутствия позиционных параметров и назначения им значений по умолчанию. Ниже приводится пример такой подстановки:

```
${параметр:-слово}
```

Если параметр не определен (то есть отсутствует) или содержит пустое значение, механизм подстановки вернет значение указанного слова. Если параметр не пустой, механизм подстановки вернет значение параметра.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:-"substitute value if unset"}
substitute value if unset
[me@linuxbox ~]$ echo $foo

[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:-"substitute value if unset"}
bar
[me@linuxbox ~]$ echo $foo
bar
```

Вот еще один вариант подстановки, где вместо дефиса используется знак «равно»:

```
${параметр:=слово}
```

Если параметр не определен или содержит пустое значение, механизм подстановки вернет значение указанного слова и дополнительно присвоит его параметру. Если параметр не пустой, механизм подстановки вернет значение параметра.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:="default value if unset"}
default value if unset
[me@linuxbox ~]$ echo $foo
default value if unset
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:="default value if unset"}
bar
[me@linuxbox ~]$ echo $foo
bar
```

ПРИМЕЧАНИЕ

Таким способом нельзя присваивать значения позиционным и другим специальным параметрам.

Ниже демонстрируется форма со знаком вопроса:

```
${параметр:?слово}
```

Если параметр не определен или содержит пустое значение, механизм подстановки завершит сценарий с ошибкой и выведет значение указанного слова

в стандартный вывод ошибок. Если параметр не пустой, механизм подстановки вернет значение параметра.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:?"parameter is empty"}
bash: foo: parameter is empty
[me@linuxbox ~]$ echo $?
1
[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:?"parameter is empty"}
bar
[me@linuxbox ~]$ echo $?
0
```

Ниже демонстрируется форма со знаком «плюс»:

```
${параметр:+слово}
```

Если параметр не определен или содержит пустое значение, механизм подстановки вернет пустое значение. Если параметр не пустой, механизм подстановки вернет значение слова, но сам параметр не изменится.

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo:+"substitute value if set"}

[me@linuxbox ~]$ foo=bar
[me@linuxbox ~]$ echo ${foo:+"substitute value if set"}
substitute value if set
```

Получение имен переменных

Командная оболочка может возвращать имена переменных. Это используется в некоторых экзотических ситуациях.

```
${!префикс*}
${!префикс@}
```

Эти две формы подстановки возвращают имена существующих переменных, начинающиеся с указанного префикса. Согласно документации bash, обе формы действуют совершенно одинаково. Следующая команда выводит список всех переменных окружения с именами, начинающимися с BASH:

```
[me@linuxbox ~]$ echo ${!BASH*}
BASH BASH_ARGC BASH_ARGV BASH_COMMAND BASH_COMPLETION BASH_COMPLETION_DIR BASH_
LINENO BASH_SOURCE BASH_SUBSHELL BASH_VERSINFO BASH_VERSION
```

Операции со строками

Существует множество форм подстановки, которые можно использовать для работы со строками. Многие из них особенно хорошо подходят для операций с путями. Форма

```
${#параметр}
```

вернет длину строки, содержащуюся в указанном параметре. Обычно роль параметра играет строка, но если передать `@` или `*`, то механизм подстановки вернет число позиционных параметров.

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo "'$foo' is ${#foo} characters long."
'This string is long.' is 20 characters long.
```

Следующая форма подстановки:

```
${параметр:смещение}
${параметр:смещение:длина}
```

используется для извлечения фрагмента строки, содержащейся в параметре. Извлечение начинается с указанного смещения от начала строки и продолжается до конца строки, если не указана длина.

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo ${foo:5}
string is long.
[me@linuxbox ~]$ echo ${foo:5:6}
string
```

Если указать отрицательное смещение, его отсчет начнется с конца строки вместо начала. Обратите внимание, что отрицательному значению должен предшествовать пробел, чтобы предотвратить путаницу с формой `${параметр:-слово}`. Длина, если указана, в этом случае не должна быть меньше 0.

Если в качестве параметра передать `@`, результатом подстановки будет длина позиционных параметров, начиная с указанного смещения.

```
[me@linuxbox ~]$ foo="This string is long."
[me@linuxbox ~]$ echo ${foo: -5}
long.
[me@linuxbox ~]$ echo ${foo: -5:2}
lo
```

Следующие две формы:

```
${параметр#шаблон}
${параметр##шаблон}
```

возвращают значение параметра, удаляя из него начальную часть, определяемую указанным шаблоном.

В шаблоне допускается использовать групповые символы: например, те, что используются в подстановке путей. Эти две формы отличаются тем, что форма # удаляет кратчайшее совпадение, тогда как форма ## удаляет самое длинное совпадение.

```
[me@linuxbox ~]$ foo=file.txt.zip
[me@linuxbox ~]$ echo ${foo#*.}
txt.zip
[me@linuxbox ~]$ echo ${foo##*.}
zip
```

Следующие две формы:

```
${параметр%шаблон}
${параметр%%шаблон}
```

действуют так же, как формы # и ##, представленные выше, но удаляют текст с конца строки, содержащейся в параметре.

```
[me@linuxbox ~]$ foo=file.txt.zip
[me@linuxbox ~]$ echo ${foo%.*}
file.txt
[me@linuxbox ~]$ echo ${foo%%.*}
file
```

Следующие формы:

```
${параметр/шаблон/строка}
${параметр//шаблон/строка}
${параметр/#шаблон/строка}
${параметр/%шаблон/строка}
```

выполняют поиск с заменой в содержимом указанного параметра. Если в параметре будет найдено совпадение с шаблоном, который может содержать групповые символы, это совпадение будет заменено содержимым указанной строки. Первая форма заменит только первое совпадение с шаблоном. Форма // заменит

все найденные совпадения. Форма `/#` выполняет замену, только если совпадение с шаблоном найдено в самом начале строки, а форма `/%` выполняет замену, только если совпадение найдено в конце строки. Часть `/строка` можно опустить, и тогда совпавший фрагмент будет удален.

```
[me@linuxbox ~]$ foo=JPG.JPG
[me@linuxbox ~]$ echo ${foo/JPG/jpg}
jpg.JPG
[me@linuxbox ~]$ echo ${foo//JPG/jpg}
jpg.jpg
[me@linuxbox ~]$ echo ${foo/#JPG/jpg}
jpg.JPG
[me@linuxbox ~]$ echo ${foo/%JPG/jpg}
JPG.jpg
```

Механизм подстановки параметров — ценный инструмент. Его возможности для работы со строками можно использовать вместо других широко используемых команд, таких как `sed` и `cut`. Применение механизма подстановки способствует увеличению производительности сценария за счет отсутствия необходимости выполнять внешние программы. Например, изменим программу `longest-word` из предыдущей главы, задействовав подстановку параметра `${#j}` взамен подстановки команды `$(echo $j | wc -c)`, которая к тому же выполняется в подоболочке:

```
#!/bin/bash

# longest-word3 : поиск самой длинной строки в файле

for i; do
      if [[ -r "$i" ]]; then
            max_word=
            max_len=
            for j in $(strings $i); do
                  len="${#j}"
                  if (( len > max_len )); then
                        max_len="$len"
                        max_word="$j"
                  fi
            done
            echo "$i: '$max_word' ($max_len characters)"
      fi
      shift
done
```

Далее, сравним эффективность двух версий с помощью команды `time`:

```
[me@linuxbox ~]$ time longest-word2 dirlist-usr-bin.txt
dirlist-usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38 characters)

real 0m3.618s
user 0m1.544s
sys 0m1.768s
[me@linuxbox ~]$ time longest-word3 dirlist-usr-bin.txt
dirlist-usr-bin.txt: 'scrollkeeper-get-extended-content-list' (38 characters)

real 0m0.060s
user 0m0.056s
sys 0m0.008s
```

Первоначальной версии потребовалось 3,618 секунды, чтобы просканировать текстовый файл, тогда как новой версии, использующей механизм подстановки параметров, понадобилось всего 0,06 секунды — весьма существенное улучшение.

Преобразование регистра символов

`bash` поддерживает четыре подстановки параметров и два варианта команды `declare` для преобразования регистра символов в строках.

Где может пригодиться возможность преобразования регистра символов? Помимо очевидной эстетической ценности, она играет важную роль в программировании. Рассмотрим случай поиска в базе данных. Представьте, что пользователь ввел строку и мы должны найти ее в базе данных. Пользователь может ввести значение только заглавными или только строчными буквами или их комбинацией. Разумеется, мы не можем позволить себе хранить в базе данных все возможные варианты написания всех строк с заглавными и строчными буквами. Как же быть?

Подобные проблемы часто решаются путем *нормализации* пользовательского ввода. То есть его преобразованием в стандартную форму перед поиском в базе данных. Для этого можно преобразовать все символы во вводе пользователя в нижний или верхний регистр и аналогичным образом нормализовать все записи в базе данных.

Для нормализации строк — приведения их к верхнему или нижнему регистру — можно использовать команду `declare`. С помощью `declare` мы сможем гарантировать, что переменная всегда будет содержать строки в желаемом формате, независимо от их первоначального вида.

```
#!/bin/bash

# ul-declare: демонстрация преобразования регистра символов с использованием
declare

declare -u upper
declare -l lower

if [[ $1 ]]; then
      upper="$1"
      lower="$1"
      echo "$upper"
      echo "$lower"
fi
```

В предыдущем сценарии мы использовали `declare` для создания двух переменных, `upper` и `lower`. Затем мы присвоили им значение первого аргумента командной строки (позиционного параметра `$1`) и вывели их на экран.

```
[me@linuxbox ~]$ ul-declare aBc
ABC
abc
```

Как видите, аргумент командной строки (`aBc`) был нормализован.

Кроме команды `declare` поддерживаются также четыре подстановки параметров, выполняющие преобразование символов в верхний/нижний регистр. Они перечислены в табл. 34.1.

Таблица 34.1. Операции подстановки параметров, выполняющие преобразование регистра символов

Формат	Результат
`${параметр,,шаблон}`	Возвращает значение указанного параметра после преобразования всех символов в нижний регистр. Необязательный `шаблон` можно использовать для ограничения символов, подлежащих преобразованию (например `[A-F]`). Подробное описание шаблонов можно найти в странице справочного руководства (man) для команды `bash`
`${параметр,шаблон}`	Возвращает значение указанного параметра после преобразования в нижний регистр только первого символа
`${параметр^^шаблон}`	Возвращает значение указанного параметра после преобразования всех символов в верхний регистр
`${параметр^шаблон}`	Возвращает значение указанного параметра после преобразования в верхний регистр только первого символа

Работу этих операций подстановки демонстрирует следующий сценарий:

```
#!/bin/bash

# ul-param: демонстрация преобразования регистра символов
            с использованием подстановки параметров

if [[ "$1" ]]; then
      echo "${1,,}"
      echo "${1,}"
      echo "${1^^}"
      echo "${1^}"
fi
```

Результат выполнения этого сценария:

```
[me@linuxbox ~]$ ul-param aBc
abc
aBc
ABC
ABc
```

Здесь мы снова обрабатываем первый аргумент командной строки и выводим результаты выполнения четырех поддерживаемых вариантов подстановки параметров. Сценарий использует первый позиционный параметр, но вообще в операцию подстановки можно передать любую строку, переменную или строковое выражение.

Вычисление и подстановка арифметических выражений

В главе 7 мы видели, как работает механизм подстановки результатов арифметических выражений. Он используется для выполнения разных арифметических операций с целыми числами. Ниже приводится его базовый синтаксис

```
$((выражение))
```

где *выражение* — это любое допустимое арифметическое выражение.

Этот вид подстановки тесно связан с составной командой `(( ))`, использовавшейся в главе 27 для вычисления арифметических выражений (оценки истинности).

В предыдущих главах мы видели некоторые наиболее типичные выражения и операторы, а здесь рассмотрим более полный их список.

Основание системы счисления

В главе 9 мы познакомились с восьмеричными (в системе счисления с основанием 8) и шестнадцатеричными (в системе счисления с основанием 16) числами. В арифметических выражениях командная оболочка позволяет использовать целочисленные константы в системах счисления с любым основанием. В табл. 34.2 показаны формы записи чисел с указанием основания системы счисления.

Таблица 34.2. Определение основания системы счисления

Форма записи	Описание
`число`	По умолчанию числа без упоминания системы счисления интерпретируются как десятичные числа (в системе счисления с основанием 10)
`0число`	В арифметических выражениях числа, начинающиеся с нуля, интерпретируются как восьмеричные (в системе счисления с основанием 8)
`0xчисло`	Форма записи шестнадцатеричных чисел
`основание#число`	Число в системе счисления с указанным основанием

Несколько примеров:

```
[me@linuxbox ~]$ echo $((0xff))
255
[me@linuxbox ~]$ echo $((2#11111111))
255
```

В этих примерах выводится значение шестнадцатеричного числа `ff` (наибольшее двухзначное число) и наибольшее восьмизначное двоичное число (в системе счисления с основанием 2).

Унарные операторы

Оболочка поддерживает два унарных оператора, `+` и `-`, используемых для обозначения положительных и отрицательных чисел соответственно.

Простая арифметика

В табл. 34.3 перечислены обычные арифметические операторы.

Таблица 34.3. Арифметические операторы

Оператор	Описание
+	Сложение
-	Вычитание
*	Умножение
/	Целочисленное деление
**	Степень числа
%	Деление по модулю (остаток от целочисленного деления)

Действия большинства из перечисленных операторов не вызывают вопросов, кроме целочисленного деления и деления по модулю, которые требуют дополнительного рассмотрения.

Поскольку оболочка поддерживает только арифметические операции с целыми числами, результатом деления всегда будет целое число:

```
[me@linuxbox ~]$ echo $(( 5 / 2 ))
2
```

Это обстоятельство увеличивает важность операции определения остатка от деления:

```
[me@linuxbox ~]$ echo $(( 5 % 2 ))
1
```

Используя операторы деления и деления по модулю, можно определить, что деление 5 на 2 дает в результате 2 с остатком 1.

Вычисление остатка от деления удобно использовать в циклах. Это позволяет выполнять в цикле определенные операции с заданным интервалом. В примере ниже выводится строка чисел, в которой выделяются числа, кратные 5:

```
#!/bin/bash

# modulo : демонстрация оператора деления по модулю

for ((i = 0; i <= 20; i = i + 1)); do
      remainder=$((i % 5))
      if (( remainder == 0 )); then
            printf "<%d> " "$i"
      else
```

```
            printf "%d " "$i"
      fi
done
printf "\n"
```

Запустив этот сценарий, вы получите следующий результат:

```
[me@linuxbox ~]$ modulo
<0> 1 2 3 4 <5> 6 7 8 9 <10> 11 12 13 14 <15> 16 17 18 19 <20>
```

Присваивание

Хотя на данном этапе это не очевидно, тем не менее арифметические выражения могут выполнять операцию присваивания. Мы уже выполняли присваивание много раз, хотя и в других контекстах. Каждый раз, передавая переменной число, мы выполняем присваивание. То же самое можно делать в арифметических выражениях:

```
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo $foo
[me@linuxbox ~]$ if (( foo = 5 ));then echo "It is true."; fi
It is true.
[me@linuxbox ~]$ echo $foo
5
```

В примере выше мы сначала присвоили переменной `foo` пустое значение и проверили, что она действительно получила пустое значение. Далее выполнили команду `if` с составной командой `(( foo = 5 ))`. Эта команда имеет два интересных аспекта: (1) она присваивает значение 5 переменной `foo` и (2) оценивает ее значение как истинное, потому что присваивание прошло успешно.

ПРИМЕЧАНИЕ

Важно запомнить значение оператора `=` в примере выше. Одиночный знак `=` выполняет присваивание: выражение `foo = 5` говорит: «Сделать значение переменной `foo` равным 5». Двойной знак `==` определяет эквивалентность: выражение `foo == 5` говорит: «Переменная foo равна 5?» Это обстоятельство может вызывать путаницу, потому что команда `test` интерпретирует одиночный знак `=` как оператор сравнения строк. Это еще одна причина предпочесть более современные составные команды `[[ ]]` и `(( ))` вместо `test`.

В дополнение к оператору `=` командная оболочка поддерживает еще несколько очень полезных операторов присваивания, перечисленных в табл. 34.4.

Таблица 34.4. Операторы присваивания

Форма записи	Описание
`параметр = значение`	Простое присваивание. Присваивает указанное значение указанному параметру
`параметр += значение`	Присваивание со сложением. Эквивалентно выражению `параметр = параметр + значение`
`параметр -= значение`	Присваивание с вычитанием. Эквивалентно выражению `параметр = параметр - значение`
`параметр *= значение`	Присваивание с умножением. Эквивалентно выражению `параметр = параметр × значение`
`параметр /= значение`	Присваивание с целочисленным делением. Эквивалентно выражению `параметр = параметр ÷ значение`
`параметр %= значение`	Присваивание с делением по модулю. Эквивалентно выражению `параметр = параметр % значение`
`параметр++`	Постинкремент переменной. Эквивалентно выражению `параметр = параметр + 1`. (Но см. пояснение ниже.)
`параметр--`	Постдекремент переменной. Эквивалентно выражению `параметр = параметр - 1`
`++параметр`	Преинкремент переменной. Эквивалентно выражению `параметр = параметр + 1`
`--параметр`	Предекремент переменной. Эквивалентно выражению `параметр = параметр - 1`

Эти операторы присваивания обеспечивают удобный и компактный способ записи многих арифметических вычислений. Особый интерес представляют операторы инкремента (`++`) и декремента (`--`), они увеличивают или уменьшают значение своего параметра на 1. Эти операторы заимствованы из языка программирования C и внедрены в несколько других языков программирования, включая `bash`.

Операторы инкремента и декремента могут находиться перед параметром или после него. Хотя в обоих случаях они увеличивают или уменьшают значение параметра на 1, тем не менее их местоположение играет важную роль. Если оператор помещается перед параметром, сначала выполняется операция инкремента (или декремента) и только потом возвращается измененное значение параметра. Если оператор помещается за параметром, операция выполняется *после*

возврата значения. Такое поведение может показаться странным, но оно реализовано с умыслом. Взгляните на следующий пример:

```
[me@linuxbox ~]$ foo=1
[me@linuxbox ~]$ echo $((foo++))
1
[me@linuxbox ~]$ echo $foo
2
```

Если присвоить переменной `foo` значение 1 и затем увеличить ее значение с помощью оператора `++`, следующего за именем переменной, выражение вернет прежнее значение 1 переменной `foo`. Однако если вывести значение переменной второй раз, мы увидим увеличенное значение. Если поместить оператор `++` перед параметром, мы получим более ожидаемый результат:

```
[me@linuxbox ~]$ foo=1
[me@linuxbox ~]$ echo $((++foo))
2
[me@linuxbox ~]$ echo $foo
2
```

Для большинства приложений на языке командной оболочки более полезным будет префиксный оператор.

Операторы `++` и `--` часто используются совместно с циклами. Внесем некоторые улучшения в сценарий, демонстрирующий применение оператора деления по модулю, чтобы немного сократить его:

```
#!/bin/bash

# modulo2 : демонстрация оператора деления по модулю

for ((i = 0; i <= 20; ++i )); do
      if (((i % 5) == 0 )); then
            printf "<%d> " "$i"
      else
            printf "%d " "$i"
      fi
done
printf "\n"
```

Битовые операции

Командной оболочкой поддерживается класс операторов, которые манипулируют числами не совсем обычным способом. Эти операторы действуют на уровне битов.

Они применяются для выполнения некоторых низкоуровневых операций, часто связанных с установкой или чтением битовых флагов. Описание битовых операторов приводится в табл. 34.5.

Таблица 34.5. Битовые операторы

Оператор	Описание
~	Поразрядное отрицание. Изменяет значения всех битов в числе на противоположные
<<	Поразрядный сдвиг влево. Сдвигает все биты в числе на один разряд влево
>>	Поразрядный сдвиг вправо. Сдвигает все биты в числе на один разряд вправо
&	Поразрядная операция И (AND). Выполняет операцию И над всеми битами двух чисел
|	Поразрядная операция ИЛИ (OR). Выполняет операцию ИЛИ над всеми битами двух чисел
^	Поразрядная операция ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR). Выполняет операцию ИСКЛЮЧАЮЩЕЕ ИЛИ над всеми битами двух чисел

Обратите внимание, что для всех битовых операторов, кроме поразрядного отрицания, существуют соответствующие операторы присваивания (например, `<<=`).

Ниже показано, как с использованием оператора поразрядного сдвига влево вывести список степеней 2:

```
[me@linuxbox ~]$ for ((i=0;i<8;++i)); do echo $((1<<i)); done
1
2
4
8
16
32
64
128
```

Логические операторы

Как мы узнали в главе 27, составная команда `(( ))` поддерживает разные операторы сравнения. Однако существует еще несколько операторов, которые можно использовать для оценки. Полный список приводится в табл. 34.6.

Таблица 34.6. Операторы сравнения

<table>
<tr><th>Оператор</th><th>Описание</th></tr>
<tr><td><=</td><td>Меньше или равно</td></tr>
<tr><td>>=</td><td>Больше или равно</td></tr>
<tr><td><</td><td>Меньше</td></tr>
<tr><td>></td><td>Больше</td></tr>
<tr><td>==</td><td>Равно</td></tr>
<tr><td>!=</td><td>Не равно</td></tr>
<tr><td>&&</td><td>Логическое И (AND)</td></tr>
<tr><td>||</td><td>Логическое ИЛИ (OR)</td></tr>
<tr><td>выражение1?выражение2:выражение3</td><td>Тернарный (трехместный) оператор сравнения. Если выражение1 вернет ненулевое значение (арифметическую истину), будет выполнено выражение2, иначе — выражение3</td></tr>
</table>

При использовании логических операторов в арифметических выражениях действуют следующие правила: выражение, возвращающее 0, считается ложным, а выражение, возвращающее ненулевое значение, — истинным. Составная команда `(( ))` отображает результаты в обычные для командной оболочки коды завершения:

```
[me@linuxbox ~]$ if ((1)); then echo "true"; else echo "false"; fi
true
[me@linuxbox ~]$ if ((0)); then echo "true"; else echo "false"; fi
false
```

Самым странным из логических операторов выглядит тернарный (или трехместный) оператор. Этот оператор (заимствованный из языка программирования C) самостоятельно выполняет логическую проверку. Его можно использовать вместо инструкции `if/then/else`. Он оперирует тремя арифметическими выражениями (этот оператор не работает со строками), и если первое выражение оценивается как истинное (то есть возвращает ненулевое значение), выполняется второе выражение. Иначе выполняется третье выражение. Опробуем его в командной строке.

```
[me@linuxbox ~]$ a=0
[me@linuxbox ~]$ ((a<1?++a:--a))
[me@linuxbox ~]$ echo $a
```

```
1
[me@linuxbox ~]$ ((a<1?++a:--a))
[me@linuxbox ~]$ echo $a
0
```

Этот пример реализует переключение значения переменной. Каждый раз, когда выполняется оператор, значение переменной a переключается с 0 на 1 или обратно.

Обратите внимание, что прямое присваивание в этом операторе считается недопустимой операцией. Если попытаться выполнить присваивание, bash сообщит об ошибке:

```
[me@linuxbox ~]$ a=0
[me@linuxbox ~]$ ((a<1?a+=1:a-=1))
bash: ((: a<1?a+=1:a-=1: attempted assignment to non-variable (error token
        is "-=1")
```

Эту проблему можно решить, заключив выражения присваивания в круглые скобки:

```
[me@linuxbox ~]$ ((a<1?(a+=1):(a-=1)))
```

Далее приводится более полный пример использования арифметических операторов в сценарии, который выводит простую таблицу чисел:

```
#!/bin/bash

# arith-loop: сценарий для демонстрации арифметических операторов

finished=0
a=0
printf "a\ta**2\ta**3\n"
printf "=\t====\t====\n"

until ((finished)); do
      b=$((a**2))
      c=$((a**3))
      printf "%d\t%d\t%d\n" "$a" "$b" "$c"
      ((a<10?++a:(finished=1)))
done
```

В этом сценарии мы реализовали цикл until, проверяющий значение переменной finished. Первоначально переменной присвоено значение 0 (арифметическая ложь). Цикл продолжается, пока эта переменная не получит ненулевое значение.

Внутри цикла вычисляются квадрат и куб переменной-счетчика `a`. В конце цикла выполняется проверка значения этой переменной. Если оно меньше 10 (максимальное число итераций), она увеличивается на 1, иначе переменной `finished` присваивается значение 1, что превращает ее в арифметическую истину, и цикл завершается. Запустив сценарий, вы получите следующий результат:

```
[me@linuxbox ~]$ arith-loop
a       a**2    a**3
=       ====    ====
0       0       0
1       1       1
2       4       8
3       9       27
4       16      64
5       25      125
6       36      216
7       49      343
8       64      512
9       81      729
10      100     1000
```

bc — язык калькулятора для вычислений с произвольной точностью

Мы уже знаем, что командная оболочка поддерживает все виды арифметических вычислений с целыми числами, но как быть, если понадобится реализовать какие-нибудь вычисления из высшей математики или хотя бы просто задействовать вещественные числа? Ответ: никак. По крайней мере, средствами командной оболочки. Для подобных вычислений придется использовать внешнюю программу. Существует множество вариантов решения этой проблемы. Одно из них — использовать программы на встроенном Perl или AWK, но, к сожалению, описание этих языков выходит далеко за рамки данной книги.

Другое решение — использовать специализированную программу-калькулятор. В большинстве систем Linux имеется одна из таких программ — программа с именем `bc`.

Программа `bc` читает файл с исходным кодом на собственном языке, напоминающем язык C, и выполняет его. Сценарий на языке `bc` можно хранить в отдельном файле или передавать его на стандартный ввод программы. Язык `bc` поддерживает массу возможностей, включая переменные, циклы и функции, определяемые программистом. Мы не будем рассматривать программу `bc` во всех подробностях, а познакомимся лишь с некоторыми ее возможностями, чтобы вы могли получить

общее представление. Неплохое описание программы bc можно найти на странице справочного руководства (man).

Начнем с простого примера. Напишем сценарий на языке bc, складывающий два числа — 2 и 2:

```
/* Очень простой сценарий на языке bc */

2 + 2
```

Первая строка сценария — это комментарий. Для оформления комментариев в языке bc используется тот же синтаксис, что и в языке программирования C. Комментарии могут размещаться в нескольких строках, начинаясь с пары символов /* и заканчиваясь парой */.

Применение bc

Сохраним сценарий, приведенный выше, в файле *foo.bc*, а затем выполним его, как показано ниже:

```
[me@linuxbox ~]$ bc foo.bc
bc 1.06.94
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
4
```

Приглядевшись, можно обнаружить результат в самом низу, после сообщения с информацией об авторских правах. Вывод этого сообщения можно подавить параметром -q (quiet — безмолвно).

bc также можно использовать в интерактивном режиме:

```
[me@linuxbox ~]$ bc -q
2 + 2
4
quit
```

В интерактивном режиме мы просто вводим выражения и сразу же получаем результат. Команда quit завершает интерактивный сеанс.

Кроме того, существует возможность передать сценарий на стандартный ввод программы bc:

```
[me@linuxbox ~]$ bc < foo.bc
4
```

Эта возможность позволяет передавать сценарии с использованием встроенных документов, встроенных строк и конвейеров. Ниже приводится пример со встроенной строкой:

```
[me@linuxbox ~]$ bc <<< "2+2"
4
```

Пример сценария

В качестве практического примера сконструируем сценарий, вычисляющий сумму ежемесячных платежей по кредиту. Для передачи сценария программе `bc` в следующем примере используется встроенный документ:

```
#!/bin/bash

# loan-calc : сценарий вычисления суммы ежемесячных платежей по кредиту

PROGNAME="${0##*/}" # Для получения имени файла используется
                    # операция подстановки параметра

usage () {
      cat <<- EOF
      Usage: $PROGNAME PRINCIPAL INTEREST MONTHS

      Where:

      PRINCIPAL is the amount of the loan.
      INTEREST is the APR as a number (7% = 0.07).
      MONTHS is the length of the loan's term.

      EOF
}

if (($# != 3)); then
      usage
      exit 1
fi

principal=$1
interest=$2
months=$3

bc <<- EOF
      scale = 10
      i = $interest / 12
      p = $principal
```

```
    n = $months
    a = p * ((i * ((1 + i) ^ n)) / (((1 + i) ^ n) - 1))
    print a, "\n"
EOF
```

Запустив этот сценарий, вы получите следующие результаты:

```
[me@linuxbox ~]$ loan-calc 135000 0.0775 180
1270.7222490000
```

В этом примере вычисляется размер ежемесячных платежей по кредиту на сумму $135 000, выданную под 7,75 % годовых на 180 месяцев (15 лет). Обратите внимание на точность результата. Она определяется значением специальной переменной `scale` в сценарии на языке `bc`. Полное описание языка `bc` можно найти на справочной странице (man) для `bc`. Несмотря на то что форма записи математических выражений немного отличается от используемой в командной оболочке (`bc` больше напоминает язык C), значительная часть сценария все же выглядит достаточно понятной, учитывая все, что мы узнали к настоящему моменту.

Заключение

В этой главе мы узнали множество маленьких хитростей, которые могут пригодиться в практической работе. По мере роста опыта в создании сценариев умение эффективно работать со строками и числами обретает истинную ценность. Наш сценарий `loan-calc` показал, что даже простые сценарии могут производить некоторые действительно полезные вычисления.

Дополнительные сведения

Даже притом, что сценарий `loan-calc` работает, он далек от совершенства. В качестве самостоятельного упражнения попробуйте улучшить сценарий `loan-calc`, добавив в него следующие возможности:

- полную проверку аргументов командной строки;
- параметр командной строки, реализующий «интерактивный» режим, в котором сценарий будет запрашивать ввод суммы, процента и срока кредита;
- улучшенный формат вывода.

35 Массивы

В предыдущей главе мы научились работать в командной оболочке со строками и числами. Типы данных, которые мы рассматривали до сих пор, в компьютерных кругах известны как *скалярные переменные*, то есть переменные, способные хранить единственное значение.

В этой главе мы познакомимся еще с одной структурой данных, которая называется *массивом*, способной хранить множество значений. Массивы поддерживаются практически во всех языках программирования. Командная оболочка также поддерживает их, хотя и в несколько ограниченном виде. Но даже в этом случае они могут использоваться для решения многих задач программирования.

Что такое массивы?

Массивы — это переменные, хранящие более одного значения. Массивы организованы подобно таблице. Возьмем, к примеру, электронную таблицу. Электронная таблица действует подобно *двумерному массиву*. В ней есть строки и столбцы, и каждая отдельная ячейка имеет свой адрес, определяемый номером строки и номером столбца. Массив устроен аналогично. Массив состоит из ячеек, которые называют *элементами*, и каждый элемент содержит данные. Доступ к отдельному элементу осуществляется с использованием его адреса, который называется *индексом*.

Большинство языков программирования поддерживает *многомерные массивы*. Электронная таблица — это пример многомерного массива с двумя измерениями: ширина и высота. Многие языки поддерживают массивы с произвольным числом измерений, однако на практике чаще всего, пожалуй, используются двух- и трехмерные массивы.

Массивы в `bash` ограничены единственным измерением. Их можно рассматривать как электронные таблицы с единственным столбцом. Но даже с этим ограничением массивам можно найти массу применений. Впервые поддержка массивов появилась в `bash` версии 2. Оригинальная командная оболочка Unix `sh` вообще не поддерживала их.

Создание массива

Переменным-массивам можно давать такие же имена, что и другим переменным `bash`, и они точно так же создаются автоматически при первом обращении к ним. Например:

```
[me@linuxbox ~]$ a[1]=foo
[me@linuxbox ~]$ echo ${a[1]}
foo
```

Это пример присваивания значения элементу массива и обращения к нему. Первая команда присваивает значение `foo` элементу массива `a` с индексом 1. Вторая команда выводит значение, хранящееся в элементе с индексом 1. Использование фигурных скобок во второй команде является обязательным условием, иначе командная оболочка будет пытаться выполнить подстановку пути, опираясь на имя элемента массива.

Массив можно также создать командой `declare`:

```
[me@linuxbox ~]$ declare -a a
```

Параметр `-a` в этом примере требует от `declare` создать массив (array) с именем `a`.

Присваивание значений массиву

Значения элементам массивов можно присваивать одним из двух способов. Присваивание одиночных значений осуществляется с использованием следующего синтаксиса:

```
имя[индекс]=значение
```

где ***имя*** — это имя массива, ***индекс*** — целое число (или арифметическое выражение) больше или равное 0. Обратите внимание, что первый элемент массива имеет индекс 0, а не 1. ***значение*** — строка или целое число, присваиваемое элементу массива.

Присвоить сразу множество значений можно с использованием следующего синтаксиса:

```
имя=(значение1 значение2 ...)
```

где *имя* — это имя массива, а *значение1 значение2 ...* — значения, присваиваемые последовательным элементам массива, начиная с элемента с индексом 0. Например, если понадобится присвоить элементам массива `days` сокращенные названия дней недели, это можно сделать так:

```
[me@linuxbox ~]$ days=(Sun Mon Tue Wed Thu Fri Sat)
```

Можно присваивать значения конкретным элементам, указывая индекс для каждого значения:

```
[me@linuxbox ~]$ days=([0]=Sun [1]=Mon [2]=Tue [3]=Wed [4]=Thu [5]=Fri [6]=Sat)
```

Доступ к элементам массива

Итак, где могут пригодиться массивы? Так же как многие задачи управления данными могут решаться с применением программ электронных таблиц, массивы могут применяться для решения множества задач программирования.

Рассмотрим простой пример сбора и представления данных. Напишем сценарий, проверяющий время последнего изменения файлов в указанном каталоге. На основе полученных данных сценарий будет выводить таблицу, показывающую, сколько файлов было изменено в каждый час суток. Такой сценарий можно использовать, например, для выяснения периодов наибольшей активности системы. Сценарий с названием `hours` производит следующий результат:

```
[me@linuxbox ~]$ hours .
Hour    Files   Hour    Files
----    -----   ----    -----
00      0       12      11
01      1       13      7
02      0       14      1
03      0       15      7
04      1       16      6
05      1       17      5
06      6       18      4
07      3       19      4
08      1       20      1
09      14      21      0
10      2       22      0
```

```
11      5       23      0

Total files = 80
```

В этом примере мы запустили программу `hours`, передав ей текущий каталог для анализа. Она вывела таблицу, показывающую число файлов, изменявшихся в каждый час суток (0–23). Ниже показан код, осуществляющий вывод этой таблицы:

```
#!/bin/bash

# hours : сценарий для подсчета файлов по времени изменения

usage () {
      echo "usage: ${0##*/} directory" >&2
}

# Убедиться, что аргумент является каталогом
if [[ ! -d "$1" ]]; then
      usage
      exit 1
fi

# Инициализировать массив
for i in {0..23}; do hours[i]=0; done

# Собрать данные
for i in $(stat -c %y "$1"/* | cut -c 12-13); do
      j="${i#0}"
      ((++hours[j]))
      ((++count))
done

# Вывести данные
echo -e "Hour\tFiles\tHour\tFiles"
echo -e "----\t-----\t----\t-----"
for i in {0..11}; do
      j=$((i + 12))
      printf "%02d\t%d\t%02d\t%d\n" \
            "$i" \
            "${hours[i]}" \
            "$j" \
            "${hours[j]}"
done
printf "\nTotal files = %d\n" $count
```

Сценарий состоит из одной функции (`usage`) и основного тела с четырьмя разделами. В первом разделе проверяется, является ли аргумент командной строки

именем каталога. Если нет, сценарий выводит сообщение с информацией о порядке использования и завершается.

Второй раздел инициализирует массив `hours`. Для этого каждому элементу массива присваивается значение 0. Массивы не требуют специальной инициализации перед использованием, но нашему сценарию важно, чтобы в массиве не оставалось элементов с пустыми значениями. Обратите внимание на необычный способ организации цикла. Используя подстановку в фигурных скобках ({0..23}), мы смогли без труда сгенерировать последовательность слов для команды `for`.

Следующий раздел осуществляет сбор данных, вызывая программу `stat` для каждого файла в каталоге. С помощью `cut` из результата извлекается двузначный час. Внутри цикла выполняется удаление ведущих нулей из поля с часом, потому что иначе командная оболочка попытается (и, в конечном счете, потерпит неудачу) интерпретировать значения с 00 по 09 как восьмеричные числа (см. табл. 34.1). Далее сценарий увеличивает на единицу значение элемента массива, соответствующего полученному часу дня. Наконец, будет увеличен счетчик (`count`), хранящий общее число файлов в каталоге.

Последний раздел в сценарии выводит содержимое массива. Сначала выводится пара строк заголовка, а затем начинает выполняться цикл, осуществляющий вывод в четыре колонки. В заключение выводится общее число файлов.

Операции с массивами

Массивы поддерживают множество типовых операций, таких как удаление массивов, определение их размеров, сортировка и др., которым можно найти много вариантов применения в сценариях.

Вывод содержимого всего массива

Для доступа к каждому элементу массива используются индексы `*` и `@`. Так же как и в случае с позиционными параметрами, индекс `@` имеет большую практическую ценность. Например:

```
[me@linuxbox ~]$ animals=("a dog" "a cat" "a fish")
[me@linuxbox ~]$ for i in ${animals[*]}; do echo $i; done
a
dog
a
cat
```

```
a
fish
[me@linuxbox ~]$ for i in ${animals[@]}; do echo $i; done
a
dog
a
cat
a
fish
[me@linuxbox ~]$ for i in "${animals[*]}"; do echo $i; done
a dog a cat a fish
[me@linuxbox ~]$ for i in "${animals[@]}"; do echo $i; done
a dog
a cat
a fish
```

Мы создали массив `animals` и сохранили в нем три строки по два слова в каждой. Затем выполнили четыре цикла, чтобы посмотреть, как выполняется разбиение содержимого массива на слова. Инструкции `${animals[*]}` и `${animals[@]}` действуют идентично, если они не заключены в кавычки. Индекс * возвращает содержимое массива, разбитое на отдельные слова, тогда как индекс `@` возвращает три «слова», соответствующие «истинному» содержимому массива.

Определение числа элементов в массиве

Определить число элементов в массиве, так же как длину строки, можно с помощью механизма подстановки параметров. Например:

```
[me@linuxbox ~]$ a[100]=foo
[me@linuxbox ~]$ echo ${#a[@]} # число элементов в массиве
1
[me@linuxbox ~]$ echo ${#a[100]} # длина элемента с индексом 100
3
```

Мы создали массив `a` и записали строку `foo` в элемент с индексом 100. Далее с помощью механизма подстановки параметров мы определили длину массива, используя при этом форму записи индекса `@`. Затем определили длину элемента с индексом 100, содержащего строку `foo`. Обратите внимание, что даже притом, что мы присвоили строку элементу с индексом 100, `bash` сообщает, что в массиве имеется только один элемент. Такое поведение необычно для тех языков, в которых неиспользуемые элементы массива (элементы с индексами 0–99) были бы инициализированы пустыми значениями и учитывались бы при определении размера массива. В `bash` элемент массива существует, только если ему присвоено значение, независимо от индекса.

Поиск используемых индексов

Так как `bash` позволяет создавать разреженные массивы путем присваивания значений отдельным элементам, иногда требуется определить, какие элементы действительно существуют. Это можно сделать с помощью механизма подстановки параметров, как показано ниже:

```
${!массив[*]}
${!массив[@]}
```

где *массив* — это имя переменной-массива. Как и в других случаях использования * и `@` в операциях подстановки, форма `@`, заключенная в кавычки, оказывается наиболее полезной, так как выполняет подстановку нераздробленных значений элементов:

```
[me@linuxbox ~]$ foo=([2]=a [4]=b [6]=c)
[me@linuxbox ~]$ for i in "${foo[@]}"; do echo $i; done
a
b
c
[me@linuxbox ~]$ for i in "${!foo[@]}"; do echo $i; done
2
4
6
```

Добавление элементов в конец массива

Знание количества элементов в массиве не поможет, если понадобится добавить значения в конец массива, потому что значения, возвращаемые индексами * и `@`, не сообщают наибольший занятый индекс в массиве. К счастью, командная оболочка предоставляет собственное решение. Оператор присваивания `+=` автоматически добавляет значения в конец массива. Ниже мы записали три значения в массив, а затем добавили в конец еще три.

```
[me@linuxbox ~]$ foo=(a b c)
[me@linuxbox ~]$ echo ${foo[@]}
a b c
[me@linuxbox ~]$ foo+=(d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
```

Сортировка массива

Так же как и при работе с электронными таблицами, при работе с массивами часто возникает необходимость отсортировать значения. Командная оболочка не имеет прямой поддержки операции сортировки, но ее нетрудно реализовать самому:

```
#!/bin/bash

# array-sort : сортировка массива

a=(f e d c b a)

echo "Original array: ${a[@]}"
a_sorted=($(for i in "${a[@]}"; do echo $i; done | sort))
echo "Sorted array: ${a_sorted[@]}"
```

Если запустить этот сценарий, он выведет следующее:

```
[me@linuxbox ~]$ array-sort
Original array: f e d c b a
Sorted array: a b c d e f
```

Сценарий копирует содержимое исходного массива (a) во второй массив (a_sorted), выполняя трюк с подстановкой команды. Этот простой прием можно использовать для выполнения самых разных операций с массивами, просто изменяя состав конвейера.

Удаление массива

Удалить массив можно с помощью команды unset:

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
[me@linuxbox ~]$ unset foo
[me@linuxbox ~]$ echo ${foo[@]}

[me@linuxbox ~]$
```

Командой unset можно также удалить единственный элемент массива:

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
[me@linuxbox ~]$ unset 'foo[2]'
[me@linuxbox ~]$ echo ${foo[@]}
a b d e f
```

В этом примере мы удалили третий элемент массива, с индексом 2. Не забывайте, что индексация элементов массива начинается с 0, а не с 1! Отметьте также, что элемент массива нужно заключить в кавычки, чтобы предотвратить подстановку путей оболочкой.

Интересно отметить, что присваивание пустого значения массиву не уничтожает его содержимое:

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ foo=
[me@linuxbox ~]$ echo ${foo[@]}
b c d e f
```

Любая ссылка на переменную-массив без индекса возвращает элемент с индексом 0:

```
[me@linuxbox ~]$ foo=(a b c d e f)
[me@linuxbox ~]$ echo ${foo[@]}
a b c d e f
[me@linuxbox ~]$ foo=A
[me@linuxbox ~]$ echo ${foo[@]}
A b c d e f
```

Ассоциативные массивы

`bash` версии 4.0 и выше поддерживает *ассоциативные массивы*. Для индексации элементов в ассоциативных массивах используются строки, а не целые числа, как в обычных массивах. Эта возможность открывает новые подходы к управлению данными. Например, можно создать массив `colors` и в качестве индексов использовать названия цветов:

```
declare -A colors
colors["red"]="#ff0000"
colors["green"]="#00ff00"
colors["blue"]="#0000ff"
```

В отличие от обычных массивов с целочисленной индексацией, которые создаются простой ссылкой на них, ассоциативные массивы можно создать только командой `declare` с новым параметром `-A`. Доступ к элементам ассоциативного массива осуществляется во многом так же, как к элементам обычных массивов с целочисленными индексами:

```
echo ${colors["blue"]}
```

В следующей главе мы рассмотрим сценарий, использующий ассоциативные массивы для создания одного интересного отчета.

Заключение

Если на странице справочного руководства (man) для `bash` выполнить поиск слова *array*, можно найти множество его упоминаний, где описываются приемы работы с переменными-массивами. Большая часть этих описаний довольно туманна, но иногда они содержат весьма полезные сведения. Фактически массивы недостаточно широко используются в программировании на языке командной оболочки, в основном потому, что традиционные командные оболочки для Unix (такие, как `sh`) не поддерживают их. Об этом недостатке остается только сожалеть, потому что массивы очень популярны в других языках программирования и являются мощным инструментом, позволяющим решать многие задачи программирования.

Массивы и циклы по своей природе близки друг другу и часто используются вместе. Например, следующая форма цикла хорошо подходит для вычисления индексов массива:

```
for ((выражение1; выражение2; выражение3))
```

36 Экзотика

В этой главе, завершающей наше путешествие, обратимся к совершенно случайным темам. Несмотря на то что в предыдущих главах мы рассмотрели основные темы, немало особенностей bash остались неохваченными. Многие из них плохо освещены в документации и полезны в основном для тех, кто занимается интеграцией bash в дистрибутивы Linux. Но есть среди них и такие, которые, хотя и используются нечасто, могут пригодиться при решении некоторых задач программирования. Их-то мы и рассмотрим.

Группы команд и подоболочки

bash поддерживает возможность группировки команд. Воспользоваться ею можно двумя способами: либо путем *группировки команд*, либо путем применения *подоболочки*. Ниже приводятся примеры синтаксиса обоих подходов.

Группа команд:

```
{ команда1; команда2; [команда3; ...] }
```

Подоболочка:

```
(команда1; команда2; [команда3;...])
```

Группа команд заключается в фигурные скобки, а подоболочка оформляется круглыми скобками. Вот и вся разница. Однако обратите внимание, что из-за особенностей реализации группировки команд в bash фигурные скобки должны отделяться от команд пробелами и последняя команда должна завершаться точкой с запятой или символом перевода строки.

Итак, где могут пригодиться группы команд и подоболочки? Даже притом, что между ними имеются важные различия (которые будут раскрыты далее), и те и другие используются в основном для перенаправления. Рассмотрим фрагмент сценария, выполняющий перенаправление вывода множества команд:

```
ls -l > output.txt
echo "Listing of foo.txt" >> output.txt
cat foo.txt >> output.txt
```

Выглядит достаточно просто: вывод трех команд перенаправляется в файл с именем *output.txt*. Воспользовавшись приемом группировки, то же самое можно выразить более кратко:

```
{ ls -l; echo "Listing of foo.txt"; cat foo.txt; } > output.txt
```

Подоболочка используется аналогично:

```
(ls -l; echo "Listing of foo.txt"; cat foo.txt) > output.txt
```

Этот прием помог нам сэкономить силы и время на вводе текста сценария, но истинная мощь групп команд и подоболочек проявляется в конвейерах. Создавая конвейеры из команд, мы часто сталкиваемся с необходимостью объединения результатов нескольких команд в общий поток. Группы команд и подоболочки упрощают эту задачу:

```
{ ls -l; echo "Listing of foo.txt"; cat foo.txt; } | lpr
```

Здесь мы объединили вывод трех команд и передали его по конвейеру на вход команды `lpr`, чтобы напечатать отчет.

В следующем сценарии мы задействуем группы команд и рассмотрим несколько приемов, которые можно использовать в сочетании с ассоциативными массивами. Получив имя каталога, сценарий с названием `array-2` выводит список файлов, находящихся в этом каталоге, с именами пользователей и групп, владеющих файлами. В конце сценарий выводит общее количество файлов, принадлежащих каждому пользователю и каждой группе. Ниже приводится сокращенный (для экономии места) вывод сценария для каталога */usr/bin*:

```
[me@linuxbox ~]$ array-2 /usr/bin
/usr/bin/2to3-2.6                    root      root
/usr/bin/2to3                        root      root
/usr/bin/a2p                         root      root
/usr/bin/abrowser                    root      root
/usr/bin/aconnect                    root      root
/usr/bin/acpi_fakekey                root      root
```

```
/usr/bin/acpi_listen                    root      root
/usr/bin/add-apt-repository             root      root
--пропущено--
/usr/bin/zipgrep                        root      root
/usr/bin/zipinfo                        root      root
/usr/bin/zipnote                        root      root
/usr/bin/zip                            root      root
/usr/bin/zipsplit                       root      root
/usr/bin/zjsdecode                      root      root
/usr/bin/zsoelim                        root      root

File owners:
daemon    :    1 file(s)
root      : 1394 file(s)

File group owners:
crontab   :    1 file(s)
daemon    :    1 file(s)
lpadmin   :    1 file(s)
mail      :    4 file(s)
mlocate   :    1 file(s)
root      : 1380 file(s)
shadow    :    2 file(s)
ssh       :    1 file(s)
tty       :    2 file(s)
utmp      :    2 file(s)
```

Далее следует исходный код сценария (с номерами строк):

```
#!/bin/bash

# array-2: Используем массивы для подсчета файлов, принадлежащих
# разным владельцам

declare -A files file_group file_owner groups owners

if [[ ! -d "$1" ]]; then
    echo "Usage: array-2 dir" >&2
    exit 1
fi

for i in "$1"/*; do
    owner="$(stat -c %U "$i")"
    group="$(stat -c %G "$i")"
    files["$i"]="$i"
    file_owner["$i"]="$owner"
    file_group["$i"]="$group"
```

```
18      ((++owners[$owner]))
19      ((++groups[$group]))
20  done
21
22  # Вывод списка файлов
23  { for i in "${files[@]}"; do
24      printf "%-40s %-10s %-10s\n" \
25          "$i" "${file_owner["$i"]}" "${file_group["$i"]}"
26  done } | sort
27  echo
28
29  # Вывод списка владельцев
30  echo "File owners:"
31  { for i in "${!owners[@]}"; do
32      printf "%-10s: %5d file(s)\n" "$i" "${owners["$i"]}"
33  done } | sort
34  echo
35
36  # Вывод списка групп
37  echo "File group owners:"
38  { for i in "${!groups[@]}"; do
39      printf "%-10s: %5d file(s)\n" "$i" "${groups["$i"]}"
40  done } | sort
```

Рассмотрим механику работы этого сценария.

Строка 5: ассоциативные массивы должны создаваться командой `declare` с параметром `-A`. Этот сценарий создает пять таких массивов:

- `files` хранит имена файлов, найденных в каталоге, индексируется именами файлов.
- `file_group` хранит имена групп, владеющих файлами, индексируется именами файлов.
- `file_owner` хранит имена пользователей, владеющих файлами, индексируется именами файлов.
- `groups` хранит число файлов, принадлежащих каждой группе, индексируется именами групп.
- `owners` хранит число файлов, принадлежащих каждому пользователю, индексируется именами пользователей.

Строки 7–10: эти строки проверяют имя каталога, переданное в позиционном параметре. Если каталог с указанным именем не существует, выводится сообщение, описывающее порядок использования сценария, после чего сценарий завершается с кодом 1.

Строки 12–20: Эти строки выполняют обход файлов в каталоге. Команды `stat` в строках 13 и 14 извлекают имена пользователя и группы, владеющих файлом, которые затем сохраняются в соответствующих массивах (строки 16 и 17), с использованием имени файла в роли индекса. Само имя файла сохраняется в массиве `files` (строка 15).

Строки 18–19: количество файлов, принадлежащих пользователю и группе, увеличивается на 1.

Строки 22–27: вывод списка файлов. Вывод осуществляется с использованием подстановки параметров `"${массив[@]}"`, которая возвращает полный список элементов массива, каждый из которых интерпретируется как отдельное слово. При таком подходе не возникает проблем с именами файлов, содержащими пробелы. Отметьте также, что весь цикл заключен в фигурные скобки, формирующие группу команд. Это позволяет передать вывод, полученный в цикле, команде `sort`. Это важно, потому что операция подстановки в строке 12 возвращает имена файлов в неотсортированном порядке.

Строки 29–40: эти два цикла реализованы подобно циклу вывода списка файлов и отличаются только использованием подстановки `"${!массив[@]}"`, которая возвращает не список элементов, а список индексов массива.

Подстановка процессов

Несмотря на внешнее сходство и возможность объединения потоков для последующего перенаправления, между группами команд и подоболочками существуют важные отличия. Все команды, входящие в группу, выполняются в текущей оболочке, подоболочка (как можно догадаться из названия) выполняет свои команды в дочерней копии текущей командной оболочки. Это означает, что в момент запуска подоболочки создается копия окружения текущей оболочки и передается новому экземпляру оболочки. Когда подоболочка завершается, ее копия окружения уничтожается, соответственно, теряются любые изменения в окружении подоболочки (включая значения переменных). Поэтому если нет прямой необходимости в использовании подоболочки, предпочтительнее использовать группы команд. Группы команд выполняются быстрее и требуют меньше памяти.

В главе 28 мы столкнулись с одной из проблем, характерных для подоболочек, когда выяснили, что команда `read` действует в конвейерах не так, как можно было бы ожидать. Там мы сконструировали следующий конвейер:

```
echo "foo" | read
echo $REPLY
```

после выполнения которого переменная REPLY всегда оставалась пустой, потому что команда read выполняется в подоболочке и ее копия REPLY уничтожается по ее завершении.

Так как конвейеры команд всегда выполняются в подоболочке, любые команды, присваивающие значения переменным, будут сталкиваться с этой проблемой. К счастью, командная оболочка поддерживает экзотическую форму подстановки, которая называется *подстановкой процессов* и может использоваться для преодоления указанных трудностей.

Подстановка процессов оформляется двумя способами: для процессов, отправляющих результаты в стандартный вывод:

```
<(список)
```

и для процессов, принимающих данные через стандартный ввод:

```
>(список)
```

где *список* — это список команд.

Ниже показано, как использовать подстановку процессов для решения проблемы с командой read:

```
read < <(echo "foo")
echo $REPLY
```

Подстановка процессов позволяет интерпретировать вывод подоболочки как обычный файл и осуществлять его перенаправление. Так как это форма подстановки, всегда можно узнать действительное подставляемое значение:

```
[me@linuxbox ~]$ echo <(echo "foo")
/dev/fd/63
```

Вывод результата подстановки командой echo показывает, что вывод подоболочки передается через файл с именем */dev/fd/63*.

Подстановка процессов часто используется в циклах, содержащих команду read. Ниже приводится пример использования read в цикле, обрабатывающем список файлов в каталоге, созданном подоболочкой:

```
#!/bin/bash

# pro-sub : демонстрация подстановки процессов

while read attr links owner group size date time filename; do
```

```
    cat <<- EOF
        Filename:   $filename
        Size:       $size
        Owner:      $owner
        Group:      $group
        Modified:   $date $time
        Links:      $links
        Attributes: $attr
EOF
done < <(ls -l | tail -n +2)
```

Цикл выполняет `read` для каждой строки в списке с содержимым каталога. Сам список создается последней строкой в сценарии. Здесь вывод подоболочки перенаправляется на стандартный ввод цикла с помощью подстановки процесса. Команда `tail` включена в конвейер, чтобы устранить первую строку в списке, которая не нужна.

Этот сценарий выведет примерно следующее:

```
[me@linuxbox ~]$ pro_sub | head -n 20
Filename:   addresses.ldif
Size:       14540
Owner:      me
Group:      me
Modified:   2009-04-02 11:12
Links:      1
Attributes: -rw-r--r--

Filename:   bin
Size:       4096
Owner:      me
Group:      me
Modified:   2009-07-10 07:31
Links:      2
Attributes: drwxr-xr-x

Filename:   bookmarks.html
Size:       394213
Owner:      me
Group:      me
```

Ловушки

В главе 10 мы узнали, что программы могут реагировать на сигналы. Эту возможность можно добавить и в сценарии. Ни в одном из сценариев, написанных нами до сих пор, этого не требовалось (потому что они быстро завершаются и не

создают временных файлов), но в больших и сложных сценариях процедура обработки сигналов может оказаться весьма кстати.

Проектируя большие и сложные сценарии, важно предусматривать их реакцию на неожиданный выход пользователя из системы или выключение компьютера во время их выполнения. Если возникают подобные события, всем процессам посылается сигнал. Программы, представляющие эти процессы, могут выполнять некие действия, гарантирующие корректное завершение с сохранением необходимых данных. Допустим, к примеру, что мы написали сценарий, создающий временный файл во время выполнения. При внимательном подходе к проектированию мы могли бы предусмотреть удаление этого файла по завершении сценария. Было бы неплохо также предусмотреть удаление файла в случае получения сценарием сигнала, требующего преждевременного завершения программы.

Для этой цели в `bash` поддерживается механизм, известный как *ловушка* (trap). Ловушки реализуются с применением встроенной команды с соответствующим именем `trap`. Команда `trap` имеет следующий синтаксис:

```
trap аргумент сигнал [сигнал...]
```

где ***аргумент*** — это строка, которая будет прочитана и выполнена как команда, а ***сигнал*** — идентификатор сигнала, в ответ на который будет выполнена указанная команда.

Рассмотрим простой пример:

```
#!/bin/bash

# trap-demo : простой пример обработки сигналов

trap "echo 'I am ignoring you.'" SIGINT SIGTERM

for i in {1..5}; do
      echo "Iteration $i of 5"
      sleep 5
done
```

Этот сценарий определяет ловушку, которая будет выполнять команду `echo` в ответ на сигналы `SIGINT` и `SIGTERM`, получаемые сценарием во время выполнения. Ниже показано, как выглядят попытки остановить сценарий нажатием комбинации CTRL+C:

```
[me@linuxbox ~]$ trap-demo
Iteration 1 of 5
Iteration 2 of 5
^CI am ignoring you.
```

```
Iteration 3 of 5
^CI am ignoring you.
Iteration 4 of 5
Iteration 5 of 5
```

Как видите, каждый раз, когда пользователь пытается прервать работу программы, она вместо этого выводит сообщение.

Иногда бывает непросто сформировать строку с требуемой последовательностью команд, поэтому на практике в качестве команды часто используют функции. Следующий пример демонстрирует применение разных функций для обработки разных сигналов:

```
#!/bin/bash

# trap-demo2 : простой пример обработки сигналов

exit_on_signal_SIGINT () {
      echo "Script interrupted." 2>&1
      exit 0
}

exit_on_signal_SIGTERM () {
      echo "Script terminated." 2>&1
      exit 0
}

trap exit_on_signal_SIGINT SIGINT
trap exit_on_signal_SIGTERM SIGTERM

for i in {1..5}; do
      echo "Iteration $i of 5"
      sleep 5
done
```

Этот сценарий дважды использует команду `trap`, настраивая ловушки для двух сигналов. В каждой ловушке используется своя функция, которая будет вызвана для обработки конкретного сигнала. Обратите внимание на включение команды **`exit`** в обе функции обработки сигналов. Без этого сценарий продолжил бы выполняться после завершения функции.

Если во время выполнения этого сценария пользователь нажмет комбинацию CTRL+C, он увидит следующее:

```
[me@linuxbox ~]$ trap-demo2
Iteration 1 of 5
Iteration 2 of 5
^CScript interrupted.
```

ВРЕМЕННЫЕ ФАЙЛЫ

Одним из побудительных мотивов включения обработчиков сигналов в сценарии является необходимость удаления временных файлов, которые сценарии могут создавать для хранения промежуточных результатов. Выбор имен для временных файлов — целое искусство. Традиционно программы в Unix-подобных системах создают свои временные файлы в каталоге */tmp*, общем для всех и предназначенном именно для таких файлов. Однако из-за того, что каталог является общим, возникает проблема безопасности, особенно остро проявляющаяся в программах, действующих с привилегиями суперпользователя. Помимо очевидной необходимости установки соответствующих разрешений для файлов, которые могут быть доступны всем пользователям в системе, важно также давать временным файлам непредсказуемые имена. Это поможет избежать атак вида *гонка за временными файлами* (temp race attack). Ниже показан один из способов создания непредсказуемого (но все еще осмысленного) имени:

```
tempfile=/tmp/$(basename $0).$$.$RANDOM
```

Эта команда сконструирует имя файла из имени программы, идентификатора процесса (PID) и случайного целого числа. Но имейте в виду, что переменная командной оболочки `$RANDOM` возвращает значения только из диапазона от 1 до 32 767, не очень большого по компьютерным меркам, поэтому единственного экземпляра переменной недостаточно, чтобы противостоять заинтересованному злоумышленнику.

Лучший результат дает программа `mktemp` (не путайте с функцией `mktemp` из стандартной библиотеки) — она автоматически выбирает имя и создает временный файл. Программа `mktemp` принимает аргумент с шаблоном, на основе которого конструирует имя файла. Шаблон должен включать последовательность символов `X`, которые будут заменены соответствующим числом случайных букв и цифр. Чем длиннее последовательность из символов `X`, тем длиннее последовательность случайных символов. Например:

```
tempfile=$(mktemp /tmp/foobar.$$.XXXXXXXXXX)
```

Эта команда создаст временный файл и сохранит его имя в переменной `tempfile`. Символы `X` в шаблоне будут заменены случайными буквами и цифрами, соответственно, окончательное имя файла (которое в данном примере включает также значение специального параметра `$$`, возвращающего идентификатор процесса) может выглядеть, например, так:

```
/tmp/foobar.6593.UOZuvM6654
```

В сценариях, предназначенных для запуска рядовыми пользователями, разумнее отказаться от использования каталога */tmp* и создать каталог для временных файлов в домашнем каталоге пользователя, например, так:

```
[[ -d $HOME/tmp ]] || mkdir $HOME/tmp
```

Асинхронное выполнение с командой wait

Иногда возникает необходимость решать одновременно несколько задач. Мы знаем, что все современные операционные системы, даже те, которые не являются многопользовательскими, поддерживают многозадачность. Сценарии тоже можно конструировать так, что они будут действовать в многозадачном режиме.

Обычно такие сценарии запускают один или несколько дочерних сценариев, решающих вспомогательные задачи, пока родительский сценарий продолжает выполнять основной алгоритм. Однако когда таким способом запускается целая серия сценариев, возникает проблема координации действий родителя и потомков. Например, представьте, что родитель зависит от результатов работы потомка или, наоборот, и он должен дождаться, пока другой сценарий завершится, прежде чем завершиться самому.

В `bash` имеется встроенная команда, помогающая управлять асинхронным выполнением в подобных ситуациях. Команда `wait` приостанавливает выполнение родительского сценария, пока не завершится указанный процесс (то есть дочерний сценарий). Для демонстрации нам понадобятся два сценария. Ниже приводится родительский сценарий:

```
#!/bin/bash

# async-parent : пример асинхронного выполнения (родитель)

echo "Parent: starting..."

echo "Parent: launching child script..."
async-child &
pid=$!
echo "Parent: child (PID= $pid) launched."

echo "Parent: continuing..."
sleep 2

echo "Parent: pausing to wait for child to finish..."
wait $pid

echo "Parent: child is finished. Continuing..."
echo "Parent: parent is done. Exiting."
```

и дочерний сценарий:

```
#!/bin/bash

# async-child : пример асинхронного выполнения (потомок)
```

```
echo "Child: child is running..."
sleep 5
echo "Child: child is done. Exiting."
```

В этом примере дочерний сценарий тривиально прост. Фактическая работа выполняется родителем. Родительский сценарий запускает дочерний сценарий и переводит его в фоновый режим выполнения. Идентификатор дочернего процесса сохраняется в переменной `pid` путем присваивания ей значения параметра `$!`, который всегда содержит идентификатор процесса последнего задания, переведенного в фоновый режим.

Родительский сценарий продолжает работу и в конце выполняет команду `wait` с идентификатором процесса дочернего сценария. Это вызывает приостановку родительского сценария до завершения дочернего сценария, после чего родительский сценарий возобновляет работу и тут же завершается.

В ходе выполнения родительский и дочерний сценарии производят следующий вывод:

```
[me@linuxbox ~]$ async-parent
Parent: starting...
Parent: launching child script...
Parent: child (PID= 6741) launched.
Parent: continuing...
Child: child is running...
Parent: pausing to wait for child to finish...
Child: child is done. Exiting.
Parent: child is finished. Continuing...
Parent: parent is done. Exiting.
```

Именованные каналы

В большинстве Unix-подобных систем существует возможность создавать файлы специального типа, которые называются *именованными каналами* (named pipe). Именованные каналы создают соединения между двумя процессами и могут использоваться как обычные файлы. Они не пользуются большой популярностью, но знать о такой возможности и уметь пользоваться ею желательно.

В программировании широко известна архитектура под названием *клиент/сервер*, основанная на использовании *механизмов взаимодействий процессов*, таких как именованные каналы или сетевые соединения.

Наиболее широко архитектура клиент/сервер используется в веб-приложениях, где веб-браузеры взаимодействуют с веб-серверами. Веб-браузер действует

как клиент, посылая запросы серверу, в ответ на которые сервер посылает веб-страницы.

Именованные каналы имеют некоторое сходство с файлами, но на самом деле образуют буферы, действующие по принципу очереди: первым пришел, первым вышел (First-In, First-Out, FIFO). Так же как в случае с обычными (неименованными) каналами, данные записываются с одного конца канала и извлекаются из другого. С применением именованных каналов можно, например, выполнять следующие команды:

```
процесс1 > именованный_канал
```

и

```
процесс2 < именованный_канал
```

и такая пара команд будет действовать подобно конвейеру

```
процесс1 | процесс2
```

Создание именованного канала

Прежде чем использовать именованный канал, его нужно создать. Это делается с помощью команды `mkfifo`:

```
[me@linuxbox ~]$ mkfifo pipe1
[me@linuxbox ~]$ ls -l pipe1
prw-r--r-- 1 me   me    0 2018-07-17 06:41 pipe1
```

Здесь с помощью команды `mkfifo` создается именованный канал с именем `pipe1`. Командой `ls` мы исследовали созданный файл, и, как видите, первой в поле с атрибутами стоит буква `p`, сообщающая, что это именованный канал (pipe).

Использование именованных каналов

Чтобы показать, как работают именованные каналы, откроем два окна терминала (или, как вариант, выполним описанные ниже действия в двух виртуальных консолях). В первом терминале введите простую команду и перенаправьте ее вывод в именованный канал:

```
[me@linuxbox ~]$ ls -l > pipe1
```

После нажатия клавиши `ENTER` появится ощущение, что команда «зависла». Это объясняется тем, что с другого конца канала данные еще не были прочитаны.

В таких ситуациях говорят, что канал *заблокирован*. Разблокировка произойдет автоматически, как только мы подключим процесс с другого конца канала и прочитаем данные из него. Во втором окне терминала введите следующую команду:

```
[me@linuxbox ~]$ cat < pipe1
```

Во втором терминале появится список содержимого каталога, созданный в первом окне, как результат работы команды `cat`. Команда `ls` в первом окне терминала благополучно разблокируется и завершится.

Заключение

Итак, мы закончили наше путешествие. Единственное, что осталось, — это практика, практика и еще раз практика. Даже притом, что на своем пути мы охватили широкий круг вопросов, в действительности мы лишь затронули верхушку айсберга под названием «командная строка». Существуют еще тысячи программ командной строки, которые вам предстоит открыть и изучить. Начните свои исследования с каталога */usr/bin*, и вы увидите их!

У. Шоттс

Командная строка Linux. Полное руководство

2-е международное издание

Перевел с английского *А. Киселев*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*А. Руденко*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 08.2019. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 12.07.19. Формат 70×100/16. Бумага офсетная. Усл. п. л. 43,860. Тираж 1200. Заказ 0000.